Subdocuments

Subdocuments are documents embedded in other documents. In Mongoose, this means you can nest schemas in other schemas. Mongoose has two distinct notions of subdocuments: arrays of subdocuments and single nested subdocuments.

```
const childSchema = new Schema({ name: 'string' });

const parentSchema = new Schema({
  // Array of subdocuments
  children: [childSchema],
  // Single nested subdocuments. Caveat: single nested subdocs only work
  // in mongoose >= 4.2.0
  child: childSchema
});
```

Aside from code reuse, one important reason to use subdocuments is to create a path where there would otherwise not be one to allow for validation over a group of fields (e.g. dateRange.fromDate <= dateRange.toDate).

- What is a Subdocument?
- Subdocuments versus Nested Paths
- Subdocument Defaults
- Finding a Subdocument
- Adding Subdocs to Arrays
- Removing Subdocs
- Parents of Subdocs
- Alternate declaration syntax for arrays
- Alternate declaration syntax for single subdocuments

What is a Subdocument?

Subdocuments are similar to normal documents. Nested schemas can have middleware, custom validation logic, virtuals, and any other feature top-level schemas can use. The major difference is that subdocuments are not saved individually, they are saved whenever their top-level parent document is saved.

```
const Parent = mongoose.model('Parent', parentSchema);
const parent = new Parent({ children: [{ name: 'Matt' }, { name: 'Sarah' }] })
parent.children[0].name = 'Matthew';

// `parent.children[0].save()` is a no-op, it triggers middleware but
// does **not** actually save the subdocument. You need to save the parent
// doc.
parent.save(callback);
```

Subdocuments have `save` and `validate` middleware just like top-level documents. Calling `save()` on the parent document triggers the `save()` middleware for all its subdocuments, and the same for `validate()` middleware.

```
childSchema.pre('save', function (next) {
  if ('invalid' == this.name) {
    return next(new Error('#sadpanda'));
  }
  next();
});

const parent = new Parent({ children: [{ name: 'invalid' }] });
parent.save(function (err) {
  console.log(err.message) // #sadpanda
});
```

Subdocuments' `pre('save')` and `pre('validate')` middleware execute **before** the top-level document's `pre('save')` but **after** the top-level document's `pre('validate')` middleware. This is because validating before `save()` is actually a piece of built-in middleware.

```
// Below code will print out 1-4 in order
const childSchema = new mongoose.Schema({ name: 'string' });

childSchema.pre('validate', function(next) {
  console.log('2');
  next();
});

childSchema.pre('save', function(next) {
  console.log('3');
  next();
});

const parentSchema = new mongoose.Schema({
  child: childSchema
});

parentSchema.pre('validate', function(next) {
  console.log('1');
  next();
});

parentSchema.pre('save', function(next) {
  console.log('4');
  next();
});
```

Subdocuments versus Nested Paths

In Mongoose, nested paths are subtly different from subdocuments. For example, below are two schemas: one with `child` as a subdocument, and one with `child` as a nested path.

```
// Subdocument
const subdocumentSchema = new mongoose.Schema({
  child: new mongoose.Schema({ name: String, age: Number })
});
const Subdoc = mongoose.model('Subdoc', subdocumentSchema);

// Nested path
const nestedSchema = new mongoose.Schema({
  child: { name: String, age: Number }
});
const Nested = mongoose.model('Nested', nestedSchema);
```

These two schemas look similar, and the documents in MongoDB will have the same structure with both schemas. But there are a few Mongoose-specific differences:

First, instances of `Nested` never have `child === undefined`. You can always set subproperties of `child`, even if you don't set the `child` property. But instances of `Subdoc` can have `child === undefined`.

```
const doc1 = new Subdoc({});
doc1.child === undefined; // true
doc1.child.name = 'test'; // Throws TypeError: cannot read property...

const doc2 = new Nested({});
doc2.child === undefined; // false
console.log(doc2.child); // Prints 'MongooseDocument { undefined }'
doc2.child.name = 'test'; // Works
```

Subdocument Defaults

Subdocument paths are undefined by default, and Mongoose does not apply subdocument defaults unless you set the subdocument path to a non-nullish value.

```
const subdocumentSchema = new mongoose.Schema({
  child: new mongoose.Schema({
    name: String,
    age: {
      type: Number,
      default: 0
    }
  })
});
const Subdoc = mongoose.model('Subdoc', subdocumentSchema);

// Note that the `age` default has no effect, because `child`
// is `undefined`.
const doc = new Subdoc();
doc.child; // undefined
```

However, if you set `doc.child` to any object, Mongoose will apply the `age` default if necessary.

```
doc.child = {};
// Mongoose applies the `age` default:
doc.child.age; // 0
```

Mongoose applies defaults recursively, which means there's a nice workaround if you want to make sure Mongoose applies subdocument defaults: make the subdocument path default to an empty object.

```
const childSchema = new mongoose.Schema({
  name: String,
  age: {
    type: Number,
    default: 0
  }
});
const subdocumentSchema = new mongoose.Schema({
  child: {
    type: childSchema,
    default: () => ({})
  }
});
const Subdoc = mongoose.model('Subdoc', subdocumentSchema);

// Note that Mongoose sets `age` to its default value 0, because
// `child` defaults to an empty object and Mongoose applies
// defaults to that empty object.
const doc = new Subdoc();
doc.child; // { age: 0 }
```

Finding a Subdocument

Each subdocument has an `_id` by default. Mongoose document arrays have a special id method for searching a document array to find a document with a given `_id`.

```
const doc = parent.children.id(_id);
```

Adding Subdocs to Arrays

MongooseArray methods such as push, unshift, addToSet, and others cast arguments to their proper types transparently:

```
const Parent = mongoose.model('Parent');
const parent = new Parent;

// create a comment
parent.children.push({ name: 'Liesl' });
const subdoc = parent.children[0];
console.log(subdoc) // { _id: '501d86090d371bab2c0341c5', name: 'Liesl' }
subdoc.isNew; // true

parent.save(function (err) {
  if (err) return handleError(err)
  console.log('Success!');
});
```

Subdocs may also be created without adding them to the array by using the create method of MongooseArrays.

```
const newdoc = parent.children.create({ name: 'Aaron' });
```

Removing Subdocs

Each subdocument has it's own remove method. For an array subdocument, this is equivalent to calling `.pull()` on the subdocument. For a single nested subdocument, `remove()` is equivalent to setting the subdocument to `null`.

```
// Equivalent to `parent.children.pull(_id)`
parent.children.id(_id).remove();
// Equivalent to `parent.child = null`
parent.child.remove();
parent.save(function (err) {
  if (err) return handleError(err);
  console.log('the subdocs were removed');
});
```

Parents of Subdocs

Sometimes, you need to get the parent of a subdoc. You can access the parent using the `parent()` function.

```
const schema = new Schema({
  docArr: [{ name: String }],
  singleNested: new Schema({ name: String })
});
const Model = mongoose.model('Test', schema);

const doc = new Model({
  docArr: [{ name: 'foo' }],
  singleNested: { name: 'bar' }
});

doc.singleNested.parent() === doc; // true
doc.docArr[0].parent() === doc; // true
```

If you have a deeply nested subdoc, you can access the top-level document using the `ownerDocument()` function.

```
const schema = new Schema({
  level1: new Schema({
    level2: new Schema({
      test: String
    })
  })
});
const Model = mongoose.model('Test', schema);

const doc = new Model({ level1: { level2: 'test' } });

doc.level1.level2.parent() === doc; // false
doc.level1.level2.parent() === doc.level1; // true
doc.level1.level2.ownerDocument() === doc; // true
```

Alternate declaration syntax for arrays

If you create a schema with an array of objects, Mongoose will automatically convert the object to a schema for you:

```
const parentSchema = new Schema({
  children: [{ name: 'string' }]
});
// Equivalent
const parentSchema = new Schema({
  children: [new Schema({ name: 'string' })]
});
```

Alternate declaration syntax for single nested subdocuments

Unlike document arrays, Mongoose 5 does not convert an objects in schemas into nested schemas. In the below example, `nested` is a *nested path* rather than a subdocument.

```
const schema = new Schema({
  nested: {
    prop: String
  }
});
```

This leads to some surprising behavior when you attempt to define a nested path with validators or getters/setters.

```
const schema = new Schema({
  nested: {
    // Do not do this! This makes `nested` a mixed path in Mongoose 5
    type: { prop: String },
    required: true
  }
});

const schema = new Schema({
  nested: {
    // This works correctly
    type: new Schema({ prop: String }),
    required: true
  }
});
```

Next Up

Now that we've covered Subdocuments, let's take a look at querying.