Mongoose Schemas

A Mongoose Schema is a JSON-ish class that has information about properties/field types of a document. It also can store information about validation and default values, and whether a particular property is required. Schemas can contain business logic and other important information.

In other words, schemas serve as blueprints for documents. They are needed for model creation (i.e., schemas are compiled into models). So, before we can use models properly, we need to define their schemas, e.g., the book schema with the name property of string type:

```
const bookSchema = mongoose.Schema({
  name: String
})
```

Warning Mongoose ignores those properties that aren't defined in the model's schema.

Mongoose Schema supports these data types:

- `String`: a standard JavaScript/Node.js string (a sequence of characters) type
- `Number`: a standard JavaScript/Node number type up to 253 (64-bit); larger numbers with mongoose-long, Git
- `Boolean`: a standard JavaScript/Node Boolean type—true or false
- `Buffer`: a Node.js binary type (images, PDFs, archives, and so on)

- `Date`: an ISODate formatted date type, such as 2014-12-31T12:56:26.009Z
- `Array`: a standard JavaScript/Node array type
- `Schema.Types.ObjectId` a typical, MongoDB 24-character hex string of a 12-byte binary number (e.g., 52dafa354bd71b30fa12c441)
- `Schema.Types.Mixed`: any type of data (i.e., flexible free type)

Warning Mongoose does not listen to mixed-type object changes, so call `markModified()` before saving the object to make sure changes in the mixed-type field are persistent.

`ObjectId` is added automatically as a primary `_id` key if omitted in the `insert` or `save` methods; `_id` key can be used to sort documents chronologically. They are available through `Schema.Types` or `mongoose.Schema.Types` (e.g., `Schema.Types.Mixed`).

We have a great deal of flexibility in defining our document schemas—for example,

```
const ObjectId = mongoose.Schema.Types.ObjectId,
  Mixed = mongoose.Schema.Types.Mixed
const bookSchema = mongoose.Schema({
  name: String,
  created_at: Date,
  updated_at: {type: Date, default: Date.now},
  published: Boolean,
  authorId : { type: ObjectId, required: true },
    description: { type: String, default: null },
    active: {type: Boolean, default: false},
    keywords: { type: [ String ], default: [] }
    description: {
      body: String,
      image: Buffer
  },
  version: {type: Number, default: function() {return 1;}},
  notes: Mixed,
  contributors: [ObjectId]
})
```

It's possible to create and use custom types (e.g., there's a module mongoose-types) that already have the rules for the ubiquitous e-mail and URL types.

Mongoose schemas are pluggable, which means, by creating a plugin, certain functionality can be extended across all schemas of the application.

For better code organization and code re-use, in the schema, we can set up static and instance methods, apply plugins, and define hooks.

Tip For validation in Node.js, consider using the `validatior.js` and `express-validator` modules.