Custom Schema Types

Mongoose allows us to define/write getters (`get`), setters (`set`), and defaults (`default`) right in the Schema! Same goes for validate and some other useful methods.

Here's an examples of defining `set` (transfer to lower case when the value is assigned), `get` (when the number is extracted the “thousands” commas are added to it), `default` (brand new ObjectID is generated), and `validate` (checks for e-mail patterns and is triggered upon `save()`) all right in a JSON-like structure of the `Schema`:

```
const postSchema = new mongoose.Schema({
  slug: {
    type: String,
    set: function(slug) {
      return slug.toLowerCase()
    }
  },
  numberOfLikes: {
    type: Number,
    get: function(value) {
      return value.toString().replace(/\B(?=(\d{3})+(?!\d))/g, ",")
    }
  },
  posted_at: {
    type: String,
    get: function(value) {
      if (!value) return null;
      return value.toUTCString()
    }
  },
  authorId: {
    type: ObjectId,
    default: function() {
      return new mongoose.Types.ObjectId()
    }
  },
  email: {
    type: String,
    unique: true,
    validate: [
      function(email) {
        return (email.match(/[a-z0-9!#$%&'*+\/=?^_`{|}~-]+(?:\.[a-z0
      'Invalid email'
    ]
  }
  })
```

If defining custom methods in the `Schema` definition is not an option for some reason (maybe our system requires us to do it dynamically), there's another approach to amending `Schema` behavior—to use chained methods:

1. Use `Schema.path(name)` to get `SchemaType` (official docs).

2. Use `SchemaType.get(fn)` to set the getter method (official docs).

For example,

```
userSchema.path('numberOfPosts')
  .get(function(value) {
    if (value) return value
    return this.posts.length
  })
```

Path is just a fancy name for the nested field name and its parent objects, for example if we have ZIP code (`zip`) as a child of `contact.address` such as `user.contact.address.zip, the` the `contact.address.zip` is a path.