	Database Relationships	75	
	Express Router	91	

Database Relationships

→ One to few (1 to 100s)

→ One to many (1 to 1000s)

→ One to squillions (1 to 10^ns)

* One to few

→ Store the child document inside parent

```
- {
    _id: ObjectId("651d...20"),
    username: "abcd",
    addresses: [
        { location: "221 Baker Street", city: "London" },
        { location: "36 Down Town", city: "London" }  ],
    __v: 1
  }
```

→ Addresses are not more than 1000 in most of the cases and we don't use addresses individually (we use with username), so ~~we~~ no need to NEW COLLECTION just for addresses.

e.g.

```
- const { Schema } = mongoose;

  const userSchema = new Schema( {

      username: String,
      addresses: [
          {                                this is a document. So
                                           MongoDB creates id in default.
                                           To Not to create it, set the
              _id: false,                  value to false.
              location: String,
              city: String,
          } ]
  });

  const User = mongoose.model("User", userSchema);


  const addUsers = async () => {

      let user1 = new User ({

          username: "abcd",
          addresses: [ {
                          location: "221 Baker Street",
                          city: "London"
                      }]
      });

      user1.addresses.push({location: "36 DownTown", city: "London"});
      await user1.save();
  };
  addUsers();
```

* One to Many

→ store a reference to the child document inside parent

```
- {
    _id: ObjectId("651d...14"),
    name: "abcdef",
    orders: [
                    ObjectId("651d...002"),
                    ObjectId("651d...004") ],
    __v: 0
  }
```

e.g.

```
- const { Schema } = mongoose;

  const productSchema = new Schema({
        item: String,
        price: Number
  });

  const Product = mongoose.model("Product", orderSchema);

  const addOrders = async () => {

        await Product.insertMany([
                    { item: "samosa",   price: 12 },
                    { item: "chips",    price: 10 },
                    { item: "choco",    price: 5 }  ]);
  };
  addOrders();
```

```
const customerSchema = new Schema({

    name: String,
    orders: [
        {
          type: Schema.Types.ObjectId,
          ref: ~~"Orde~~ "Product"   ← collection name in which ↓ this _id exist
        }]
});

const Customer = mongoose.model("Customer", customerSchem

      addCustomer
const ~~addCustomer~~ = async () => {

    let cust1 = new Customer ({
        name: "Rahul Kumar"
    });

    let order1 = await Product.findOne({item: "Chips"});
    let order2 = await Product.findOne({item: "choco"});

    cust1.orders.push(order1);
    cust1.orders.push(order2);

    await cust1.save();
};

addCustomer();
```

* Populate

```
> db.customers.find()
[
  {
      _id: ... ,
      name: ... ,
      orders: [ ObjectId("..."), ObjectId("...") ],
      __v: 0
  }
]
```

→ As you can see in above example, only _id is stored in orders key.

→ If we want to see original document then we have to use POPULATE method.

```
- Customer.findOne({name: "Rahul Kumar"})
          .populate("orders");

    output:-   ...,
               orders: [ { _id: ...,
                           item: ...,
                           price: ... }, {...} ], ...
```

→ If we want only item key after populate then.

```
- Customer.findOne( { name: "Rahul Kumar" } )
          .populate("orders", "item");
```

* One to Squillions

→ Store reference of the parent document inside child. e.g. posts of instagram, facebook or videos of youtube, .etc.

```
- {
      _id: ObjectId("651d...a9"),
      content: "Hello1",
      likes: 7,
      user: ObjectId("651d... 33"),
      __v: 0
  },
  {
      _id: ObjectId("651d... 19"),
      content: "Hello2",
      likes: 25,
      user: ObjectId("651d ... 33"),
      __v: 0.
  }
```

```
- const userSchema = new Schema({
      username: String,
      email: String
  });
  const postSchema = new Schema({
      content: String,
      likes: Number,
      user: {
                type: Schema.Types.ObjectId,
  });           ref: "User" }
```

```
const User = mongoose.model("User", userSchema);
const Post = mongoose.model("Post", postSchema);

~~const addData new User~~
       ~~username  "rahulKumar"~~
       ~~email "rahul@abc.com"~~

const addData = async () => {

    let user1 = new User ({
           username: "rahulKumar",
           email: "rahul@abc.com"
    });

    let post1 = new Post ({
          content: "Hello1",
          likes: 7
    });

    post1.user = user1;

    await user1.save();
    await post1.save();

};

addData();
```

* 6 Rules for MongoDB Schema Design

1. Favor embedding unless there is a compelling reason not to.

2. Needing to access an object on its own is a compelling reason not to embed it.

3. Arrays should not grow without bound. If there are more than a couple of hundred documents on the "many" side, don't embed them; if there are more than a few thousand documents on the "many" side, don't use an array of objectID references. High-cardinality arrays are a compelling reason not to embed.

4. Don't be afraid of application-level joins: If you index correctly and use the projection specifier, then application-level joins are barely more expensive than server-side joins in a relational database.

5. Consider the read-to-write ratio with denormalization. A field that will mostly be read and only seldom updated is a good candidate for denormalization. If you denormalize a field that is updated frequently then extra work of finding and updating all the instances of redundant

data is likely to overhelm the savings that you get from denormalization.

6. As always with MongoDB, how you model your data depends entirely on your particular application's data access patterns. You want to structure your data to match the ways that your application queries and updates it.

* Denormalization

→ Store duplicate data

eg. We have "username" in userSchema and we can also store "username" as a key-value pair in "uses" key which is object of reference. So, if we populate "uses" key then you can see two "username" key-value pairs.

→ In this case, if we edit the value, we also have to consider where else we have stored the duplicate and edit there, too. So, we denormalization to read-only data.

* Two-Way Referencing

→ Combination of approach one to many and many to one.

users collection

```
- {
    _id: ObjectId("AAA"),
    name: "abc",
    tasks: [ ObjectID("BBB"), ObjectId("BBA")]
  }
```

tasks collection

```
- {
    _id: ObjectId("BBB"),
    description: "Write plan",
    due_date: ISODate(" 2014-04-01"),
    owner: ObjectID("AAA")
  }
```

* Handling Deletion

→ If one model stored references of other model then when deleting document of model one, we also have to delete data of the other model.

e.g. If user deletes their account then we also have to delete all of their posts.

→ We use mongoose middlewares

→ pre - run before the query is executed
→ post - run after the query is executed.

→ mongoose don't have findByIdAndDelete middleware but findByIdAndDelete automatically triggers findOneAndDelete middleware.

```
                                        ↙ middleware name
— customerSchema.post("findOneAndDelete", async(customer)=>{

    if (customer.posts.length) {

        await Post.deleteMany( {
                        _id : { $in : customer.posts }
        });
    }
});

const delCust = async () => {
 await Customer.findByIdAndDelete("65...157"); };
delCust();
```

eg. Suppose, we have listings and those listings have reviews stored in an array. Now, we want to delete one listing. So, we also have to delete reviews from Review collection.

In models/listing.js

```
- listingSchema.post("findOneAndDelete", async (listing)=>{
    if(listing) {
      await Review.deleteMany( {
            _id: { $in: listing.reviews }
      });
    }
  });
```

eg. We want to delete review so, we will delete from Review collection as well as review id from listing collection.

In app.js

```
- app.delete("/listings/:id/reviews/:reviewId",
           wrapAsync(async (req, res, next) => {

    let { id, reviewId } = req.params;

    await Listing.findByIdAnd~~Delete~~Update( id,
            { $pull : { reviews: reviewId } } );

    await Review.findByIdAndDelete(reviewId);

    res.redirect(`/listings/${~~listing~~ id}`);
}));
```

⇒ $pull This operator removes elements from an existing array for all instances of a value or values that match a specified condition.
- It searches as per our condition and pulls that whole referenceID.

Express Router

→ Express Router are a way to organize your Express application such that our primary app.js file does not become bloated.

```
const router = express.Router();
```
↑ creates a new Router object

In app.js

```
- const express = require("express");
  const app = express();
  const users = require("./routes/user.js");
  const posts = require("./routes/post.js");

  app.use("/users", users);
  app.use("/posts", posts);
```
↑ url path ↑ objects

⇒ We can change names as userRouter & postRouter

In routes/user.js

```
- const express = require("express");
  const router = express.Router();

router.get("/", (req, res) => { res.send("GET for users"); } );
router.get("/:id", (req, res) => { res.send("GET for userid"); } );
router.post("/", (req, res ) => { res.send ("POST for users"); } );
router.delete("/:id", (req, res) => { res.send ("DELETE for userid"); });
 module.exports = router;
```

* Access params from main js file

In app.js

- app.use("/listings/:id/reviews", reviews);

In routes/review.js

```
- router.post("/", ...

      const id = req.params.id;

  });
```

→ We need id of listing but it is in app.js file to access it in review.js file, we will use "mergeParams" option in reviews.js file while creating router object.

In routes/review.js

```
- const router = express.Router(
                          { mergeParams: true });
```