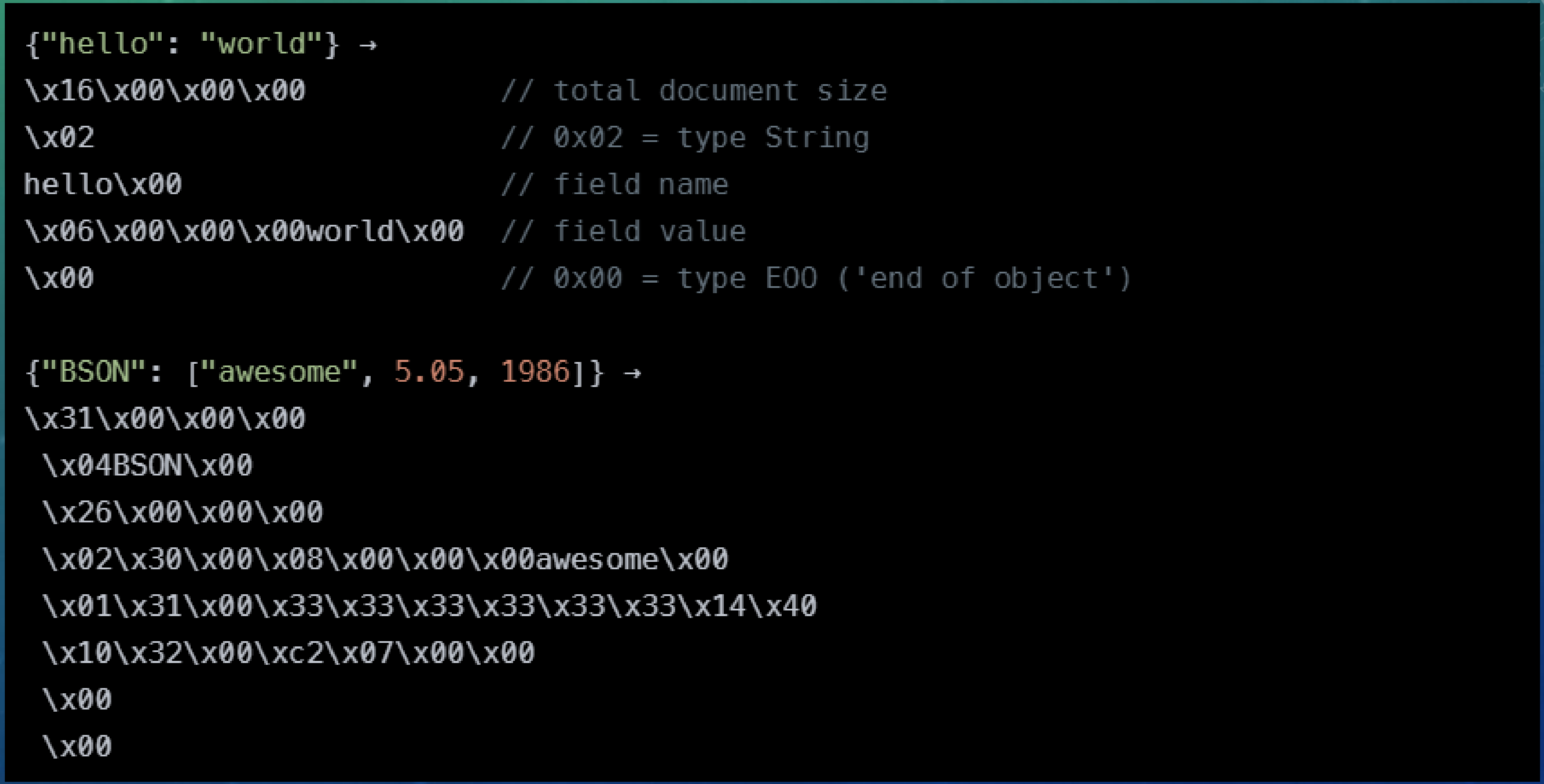
READ FOR THE STEP3: KEY FEATURES OF MONGODB

1. Data is stored in **BSON** and presented in **JSON**.
2. Server-side JavaScript is supported (JavaScript expressions and functions).
3. Document oriented database where there are no tables and no row-based data.
4. NoSQL, where there is no schema. Documents can have different structures. Documents can be embedded. This is specifically designed for horizontal scaling.
5. By default, there is a **primary key (_id)**, which is an auto-generated field for every document.
6. **Sharding** is supported which is very much essential for horizontal scaling and replication.
7. It has automatic **load balancing and fault tolerance** configurations.

READ FOR THE STEP 3
WHAT BSON LOOKS LIKE



```
{"hello": "world"} →
\x16\x00\x00\x00           // total document size
\x02                       // 0x02 = type String
hello\x00                  // field name
\x06\x00\x00\x00world\x00  // field value
\x00                       // 0x00 = type EOO ('end of object')

{"BSON": ["awesome", 5.05, 1986]} →
\x31\x00\x00\x00
 \x04BSON\x00
 \x26\x00\x00\x00
 \x02\x30\x00\x08\x00\x00\x00awesome\x00
 \x01\x31\x00\x33\x33\x33\x33\x33\x33\x14\x40
 \x10\x32\x00\xc2\x07\x00\x00
 \x00
 \x00
```

READ FOR STEP 3:
WHAT IS objectId (_id) ?

- ObjectId in the MongoDB is same as the primary key in the conventional RDBMS.
- It is by default set by MongoDB for every document that is created inside any collection.
- ObjectIds are small, unique, fast to generate and ordered.
- ObjectId values comprises of a 12 bytes hexadecimal number which is unique for every document.
- _id: ObjectId(4 bytes timestamp, 3 bytes machine id, 2 bytes process id,
- 3 bytes counter)
- "_id": ObjectId("5901832c91427cac52e9ea8f")

READ FOR THE STEP 3: WE'LL CREATE THE MODELS, AND THIS INCLUDES USING DATATYPES. MAJOR DATATYPES IN MONGODB:

1. **String**
2. Integer
3. Boolean
4. Double
5. Min/Max keys
6. Arrays
7. Timestamp
8. Object
9. Null
10. Symbol
11. Date
12. ObjectID
13. Binary data
14. Code
15. Regular expression

STEP 3

- A) Create directory olympics/model and inside it - the file Role.js, that will contain definition of the collection “roles” – but in “Mongoose” we create it through defining a schema,
  and then creating a MODEL – model is like a class. We even could add it methods.

```
const mongoose = require("mongoose");

const Role = mongoose.model(
  "Role",
  new mongoose.Schema({
    userType: {
        type: String,
        enum: ['user', 'moderator', 'admin']
    }
  })
);

module.exports = Role;
```

- B) In the file Olympics/model/User.js create the model for the “users” collection:

```
const mongoose = require('mongoose');
const Role = require('./Role')

userSchema = new mongoose.Schema({
    name: { type: String, required: true },
    email: { type: String, required: true, unique: true },
    password: { type: String, required: true },
    role: { type: mongoose.Schema.Types.ObjectId, ref: 'Role',
            required: true }
})

const User = mongoose.model('User', userSchema)
module.exports = User
```

STEP 3 CONTINUED

- C) Add one more field to User schema: “sport” and make for it the default value “Judo”
- D) Create in the directory Olympics/model also common file index.js – import there Role and User
- E) When we create first documents of the models User and Role – the collections will be created, so we want to initialize the models with several documents, for this we’ll create 2 async functions:
  initRole() and initUser()

- F) In initRole() we don’t want to create the roles again and again, so check if there already some documents exist:

```
let count = await
Role.estimatedDocumentCount();
```

-

If yes, get out of the function, if not …

STEP 3 CONTINUED

- G) We create document with userType "user" and chain (in MongoDB we say "pipe") to it function "save()" – it will be saved in the collection

```
new Role({userType: "user"}).save();
```

-
  ```
          console.log("added 'user' to
  roles collection");
  ```

- H) Do the same for the userType "moderator"

  I) Create a global variable "admin" outside the function, as we want to save the pointer to the next document:

```
admin = new Role({userType: "admin"});
```

-
  ```
          admin.save();
  ```

-
  ```
          console.log("added 'admin' to
  roles collection");
  ```

- K) In initUser() we start from checking if the user we're going to create is already exist:

-
  ```
  let YaelArad = await
  User.findOne({email:'yarad@gmail.com'}
  );
  ```

-
  ```
          console.log(`YaelArad=\n`,Yael
  Arad)
  ```

-
  ```
          if (YaelArad) {
  ```

-
  ```
              console.log('The user Yael
  Arad is already exist');
  ```

-
  ```
              return;
  ```

-
  ```
          }
  ```

- L) And we create the user:
  ```
  new User({name: "Yael Arad",
  ```

-
  ```
                   email:'yarad@gmail.co
  m',password:'123',
  ```

-
  ```
                    role:
  admin._id}).save();
  ```

-
  ```
          console.log("added 'Yael Arad'
  users collection");
  ```

STEP3 – THE END

- M) We create now the async function init() to envelop the both functions initRole() and initUser() – and we want to use “await” with initRole() to make sure that we’ve got “admin” document before we create our user
- N) Finally we fill the command to run init()
- O) In the main server.js require model/index.js and it’ll run the init()
- P) ensure that you see the new roles and the new user in your Olympics DB