P.PORTO

POLITÉCNICO
DO PORTO
ESMAD

PROGRAMAÇÃO WEB II
TSIW

SUMMARY

- 1-N and N-N relatioships with Sequelize
- Tutorials API exercise: implementation of 1:N and N:N relationships
- Authentication using JWT

Relationships in Sequelize

- Sequelize supports the standard associations: 1:1, 1:N and N:M
- Creating associations in Sequelize is done by calling one of the above functions on a model (the `source - A`), and providing another model as the first argument to the function (the `target - B`):
  - A.hasOne(B): 1:1 relationship between A and B models; adds a foreign key to the B
  - A.belongsTo(B): 1:1 relationship; add a foreign key to the A
  - A.hasMany(B) : 1:N relationship; adds a foreign key to target (B)
  - A.belongsToMany(B, { through: 'C' }): N:M association between A and B, through the junction table C

Relationships in Sequelize

- They all accept an `options` object as a second parameter
  - optional for the first three, mandatory for [belongsToMany]() containing at least the `through` property

```
const A = sequelize.define('A', /* ... */);
const B = sequelize.define('B', /* ... */);

A.hasOne(B, { /* options */ });
A.belongsTo(B, { /* options */ });
A.hasMany(B, { /* options */ });
A.belongsToMany(B, { through: 'C', /* options */ });
```

- Sequelize **automatically adds foreign keys** to the appropriate models (unless they are already present)
- For the N:M association, the junction table C is also created (unless it already exists) with the appropriate foreign keys on it
- Foreign key **constraints** are also created: all associations use CASCADE on update and SET NULL on delete, except for N:M, which also uses CASCADE on delete

Relationships in Sequelize

- Usually, the Sequelize associations are defined in pairs:
  - To create a 1:1 relationship, the hasOne and belongsTo associations are used together;
  - To create a 1:N relationship, the hasMany and belongsTo associations are used together;
  - To create a N:M relationship, two belongsToMany calls are used together
- The advantages of using these pairs instead of one single association will be discussed latter

1:1 relationships

- Create a 1:1 relationship
  - In a relational database, this will be done by establishing a foreign key in one of the tables
  - Which one? Sequelize will infer what to do from the source and target models

Setup a 1:1 relationship between models Foo and Bar

```
// since no option was passed, Sequelize adds a fooId FK column into Bar model
Foo.hasOne(Bar);
Bar.belongsTo(Foo);
```

Calling `Bar.sync()` after the above will yield the following SQL:

```
CREATE TABLE IF NOT EXISTS "foos" ( /* ... */);
CREATE TABLE IF NOT EXISTS "bars" (
        /* ... */
        "fooId" INTEGER REFERENCES "foos" ("id") ON DELETE SET NULL ON UPDATE CASCADE
);
```

Relationships in Sequelize: options

- Various options can be passed as a second parameter of the association call:
  - ➢ to configure the ON DELETE and ON UPDATE behaviors

Setup a 1:1 relationship between models Foo and Bar, such that Bar gets a `fooId` column (FK)

```
Foo.hasOne(Bar, {
  onDelete: 'SET DEFAULT', // default would be 'SET NULL'
  onUpdate: 'NO ACTION'    // default would be 'CASCADE'
}););
Bar.belongsTo(Foo);
```

Calling `Bar.sync()` after the above will yield the following SQL:

```
CREATE TABLE IF NOT EXISTS "foos" ( /* ... */);
CREATE TABLE IF NOT EXISTS "bars" (
    /* ... */
    "fooId" INTEGER REFERENCES "foos" ("id") ON DELETE SET DEFAULT ON UPDATE NO ACTION
);
```

Relationships in Sequelize: options

- Various options can be passed as a second parameter of the association call:
  - to customize the foreign key name

Setup a 1:1 relationship between models Foo and Bar, with 4 options to set FK name as myFooId

```
// Option 1: Bar gets a myFooId column (FK)
Foo.hasOne(Bar, {
  foreignKey: 'myFooId'
});
Bar.belongsTo(Foo);

// Option 2 (longer)
Foo.hasOne(Bar, {
  foreignKey: {
    name: 'myFooId'
  }
});
Bar.belongsTo(Foo);
```

```
// Option 3: FK set up in belongTo
Foo.hasOne(Bar);
Bar.belongsTo(Foo, {
  foreignKey: 'myFooId'
});

// Option 4 (longer)
Foo.hasOne(Bar);
Bar.belongsTo(Foo, {
  foreignKey: {
    name: 'myFooId'
  }
});
```

Relationships in Sequelize: options

- Various options can be passed as a second parameter of the association call:
  - ➢ to alter the default optional association to a mandatory one

Setup a 1:1 relationship between Foo and Bar models, such the `fooId` column (FK) is not allowed to be null, meaning that one Bar cannot exist without a Foo

```
Foo.hasOne(Bar, {  // a fooId column (FK) must be added to Bar
  foreignKey: {
    allowNull: false // means that one Bar cannot exist without a Foo
  }
});
```

Calling `sync()` after the above will yield the following SQL:

```
CREATE TABLE IF NOT EXISTS "foos" ( /* ... */);
CREATE TABLE IF NOT EXISTS "bars" (
    /* ... */
    "fooId" INTEGER NOT NULL REFERENCES "bars" ("id") ON DELETE RESTRICT ON UPDATE RESTRICT
);
```

1:M relationships

- Create a 1:M relationship
  - In a relational database, this will be done by establishing a foreign key in the M side

Setup a 1:M relationship between Team and Player models

```
// Sequelize knows that a teamId FK column is added to Player
Team.hasMany(Player);
Player.belongsTo(Team);
```

Calling `sync()` after the above will yield the following SQL:

```
CREATE TABLE IF NOT EXISTS "teams" ( /* ... */);
CREATE TABLE IF NOT EXISTS "players" (
    /* ... */
    "teamId" INTEGER REFERENCES "team" ("id") ON DELETE SET NULL ON UPDATE CASCADE
);
```

N:M relationships in Sequelize

- Many-To-Many associations cannot be represented by just adding one foreign key to one of the tables, as the other relationships did
  - ➢ Instead, an **extra model is needed** (and extra table in the database) which will have two foreign key columns and will keep track of the associations - the junction table is also sometimes called join table or through table

```
const Movie = sequelize.define('Movie', { name: DataTypes.STRING });
const Actor = sequelize.define('Actor', { name: DataTypes.STRING });
Movie.belongsToMany(Actor, { through: 'ActorMovies' });
Actor.belongsToMany(Movie, { through: 'ActorMovies' });
```

In N:M relationships, the string given in the **through** option of the **belongsToMany** call, will automatically create the **ActorMovies** model which will act as the junction model

```
CREATE TABLE IF NOT EXISTS "ActorMovies" (
        "createdAt" TIMESTAMP WITH TIME ZONE NOT NULL,
        "updatedAt" TIMESTAMP WITH TIME ZONE NOT NULL,
        "MovieId" INTEGER REFERENCES "Movies" ("id") ON DELETE CASCADE ON UPDATE CASCADE,
        "ActorId" INTEGER REFERENCES "Actors" ("id") ON DELETE CASCADE ON UPDATE CASCADE,
        PRIMARY KEY ("MovieId","ActorId")
);
```

N:M relationships in Sequelize

- The**junction model** can also be defined by **passing a model directly**
  - ➢ no extra model will be created automatically

```
const Movie = sequelize.define('Movie', { name: DataTypes.STRING });
const Actor = sequelize.define('Actor', { name: DataTypes.STRING });
const ActorMovies = sequelize.define('ActorMovies', {
  MovieId: {
    type: DataTypes.INTEGER,
    references: {
      model: Movie, key: 'id'
    }
  },
  ActorId: {
    type: DataTypes.INTEGER,
    references: {
      model: Actor, key: 'id'
    }
  }
});
Movie.belongsToMany(Actor, { through: 'ActorMovies' });
Actor.belongsToMany(Movie, { through: 'ActorMovies' });
```

Relationships with aliases

- When creating associations, it is possible to provide an alias, using the as option
  - ➢ This is useful if the same model is associated twice, or you want your association to be called something other than the name of the target model

Example: consider the case where users have many pictures, one of which is their profile picture. All pictures have a `userId`, but in addition the user model also has a `profilePictureId`, to be able to easily load the user's profile picture

```
User.hasMany(Picture) // all pictures now have a userId attribute, as foreign key
User.belongsTo(Picture, { as: 'ProfilePicture', constraints: false }) // all users
now have a ProfilePicture attribute, as foreign key

// MIXINS: special methods to interact between models of an association
user.getPictures() // gets all user’s pictures
user.getProfilePicture() // gets only the user’s profile picture

User.findAll({
  where: ...,
  include: [
    { model: Picture }, // load all user’s pictures
    { model: Picture, as: 'ProfilePicture' }, // load the user’s profile picture
  ]
})
```

constraints: to sync the models correctly and avoid circular dependencies between User and Picture

Sequelize: queries in associations

- For creating, updating and deleting, you can either:
  - ➢ Use the standard model queries directly

```
Foo.hasOne(Bar);
Bar.belongsTo(Foo); // Bar as FK fooID

// This creates a Bar belonging to the Foo of ID 5
Bar.create({
  name: 'My Bar',
  fooId: 5
});
```

Sequelize: queries in associations

- For creating, updating and deleting, you can either:
  - ➢ Use **mixins functions:** when an association is defined between two models, their instances gain special methods to interact with their associated counterparts
  - ➢ For example, if two models, Foo and Bar, are associated, their instances will have the following mixins available, depending on the association type:

Association	Mixins

Foo.hasOne(Bar)
Foo.belongsTo(Bar)
	fooInstance.getBar()
	fooInstance.setBar()
	fooInstance.createBar()

Foo.hasMany(Bar)
Foo.belongsToMany(Bar, { through: Baz })
	fooInstance.getBars()
	fooInstance.countBars()
	fooInstance.hasBar()
	fooInstance.hasBars()
	fooInstance.setBars()
	fooInstance.addBar()
	fooInstance.addBars()
	fooInstance.removeBar()
	fooInstance.removeBars()
	fooInstance.createBar()

Sequelize: queries in associations

- For creating, updating and deleting, you can either:
  - ➢ Use **mixins functions**

Mixins are formed by a prefix (e.g. get, add, set) concatenated with the model name

```
// for now, only Foo knows about Bar, so mixins are created for the Foo model
Foo.hasOne(Bar);
// now, also Bar knows about Foo, so also for this model mixins are created
Bar.belongsTo(Foo);

// mixins for hasOne & belongsTo associations
const foo = await Foo.create({ name: 'the-foo' });
const bar1 = await Bar.create({ name: 'some-bar' });
     console.log(await foo.getBar()); // null

await foo.setBar(bar1);
     console.log((await foo.getBar()).name); // 'some-bar'

await foo.createBar({ name: 'yet-another-bar' });
const newlyAssociatedBar = await foo.getBar();
     console.log(newlyAssociatedBar.name); // 'yet-another-bar'

await foo.setBar(null); // Un-associate
      console.log(await foo.getBar()); // null
```

Sequelize: queries in associations

- Consider a Ships and Captains models, with a 1:1 relationship between them:

```
const Ship = sequelize.define('ship', {
  name: DataTypes.TEXT,
  crewCapacity: DataTypes.INTEGER,
  amountOfSails: DataTypes.INTEGER
}, { timestamps: false });

const Captain = sequelize.define('captain', {
  name: DataTypes.TEXT,
  skillLevel: { type: DataTypes.INTEGER, validate: { min: 1, max: 10 } }
}, { timestamps: false });

// the associations below create the `captainId` foreign key in Ship, while allowing
for null values, meaning that a Ship can exist without a Captain and vice-versa
Captain.hasOne(Ship);
Ship.belongsTo(Captain);
```

Sequelize: queries in associations

- **Lazy Loading**: technique of fetching the associated data only when you really want it
  - ➢ save time and memory by only fetching it when necessary

getShip() is a **mixin function**: when an association is defined between 2 models, their instances gain special methods to interact with their associated counterparts

```
const awesomeCaptain = await Captain.findOne({
  where: { name: "Jack Sparrow" }
});

// Do stuff with the fetched captain (from the previous query)
console.log('Name:', awesomeCaptain.name);
console.log('Skill Level:', awesomeCaptain.skillLevel);

// LAZY LOADING (2nd query): now we want information about his ship!
const hisShip = await awesomeCaptain.getShip();

// Do stuff with the ship
console.log('Ship Name:', hisShip.name);
console.log('Amount of Sails:', hisShip.amountOfSails);
```

Sequelize: queries in associations

- **Eager Loading**: brings the associated data with only one query
  - ➢ when Sequelize fetches associated models, they are added to the output object as model instances
  - ➢ at the SQL level, this is a query with one or more joins
  - ➢ in Sequelize, eager loading is done by using the **include** option on a model finder query

```
const awesomeCaptain = await Captain.findOne({
  where: { name: "Jack Sparrow" },
  include: Ship // EAGER LOADING, by providing the model object
 (creates a left outer join in the query)
});

// Now the ship comes with it
console.log('Name:', awesomeCaptain.name);
console.log('Skill Level:', awesomeCaptain.skillLevel);
console.log('Ship Name:', awesomeCaptain.ship.name);
console.log('Amount of Sails:', awesomeCaptain.ship.amountOfSails);

// EAGER LOADING (providing the model name)
const ship = await Ship.findAll({ include: 'captain' })
```

OUTPUT example:

```
{
  "id": 1,
  "name": "Jack Sparrow",
  "skillLevel": "10",
  "shipId": 1,
  "ship": {
       "id": 1
       "name": "John Doe",
       "crewCapacity": "100"
       "amountOfSails": "50"
   }
}
```

Sequelize: queries in associations

- Eager Loading: if an association is aliased (using the as option) the alias must be specified when including the model
    - ➢ Consider this next example, where 1:N associations are defined between User and Task and between User and Tool models

```
const User = sequelize.define('user', { name: DataTypes.STRING }, { timestamps: false });
const Task = sequelize.define('task', { name: DataTypes.STRING }, { timestamps: false });
const Tool = sequelize.define('tool', {
  name: DataTypes.STRING,
  size: DataTypes.STRING
}, { timestamps: false });
User.hasMany(Task);
User.hasMany(Tool, { as: 'Instruments' }); //aliased association


const users = await User.findAll({ include: Task }); // #1
const users = await User.findAll({                   // #2
  include: { model: Tool, as: 'Instruments' }
});
//OR
const users = await User.findAll({                   // #2
  include: 'Instruments'
});
```

OUTPUT #1:
```
[{
 "name": "John Doe",
 "id": 1,
 "tasks": [{
  "name": "A Task",
  "id": 1,
  "userId": 1
 }]
}]
```

OUTPUT #2:
```
[{
 "name": "John Doe",
 "id": 1,
 "Instruments": [{
  "name": "Scissor",
  "id": 1,
  "userId": 1
 }]
}]
```

Sequelize: queries in associations

- Eager Loading: one can force the query to return only records which have an associated model, effectively converting the query from the default OUTER JOIN to an INNER JOIN
  - This is done with the `required: true` option

```
const users = await User.findAll({
  include: { model: Tool, as: 'Instruments', required: true }
});
```

Generated SQL:
SELECT * FROM \`users\` AS \`user\`
INNER JOIN \`tools\` AS \`Instruments\` ON \`user\`.\`id\` = \`Instruments\`.\`userId\`

Sequelize: queries in associations

- Eager Loading: one can also filter the associated model using the `where` option
  - when the where option is used inside an include, Sequelize **automatically sets the `required` option to true**. So, instead of the default OUTER JOIN, an INNER JOIN is done, returning only the parent models with at least one matching children

```
const users = await User.findAll({
  include: { model: Tool, as: 'Instruments',
      where: {
         size: {
           [Op.ne]: 'small'
         }
       } }
});
```

Generated SQL:
SELECT * FROM `users` AS `user`
INNER JOIN `tools` AS `Instruments` ON `user`.`id` = `Instruments`.`userId`
AND `Instruments`.`size` != 'small';

Sequelize: queries in associations

- Eager Loading: to obtain top-level WHERE clauses that involve nested columns, the '`$nested.column$`' syntax of Sequelize provides a way to reference nested columns
  - In SQL, the default OUTER JOIN is used

```
User.findAll({
  where: {
    '$Instruments.size$': { [Op.ne]: 'small' }
  },
  include: [{
    model: Tool,
    as: 'Instruments'
  }]
});
```

Generated SQL:
SELECT * FROM `users` AS `user`
LEFT OUTER JOIN `tools` AS `Instruments` ON `user`.`id` = `Instruments`.`userId`
WHERE `Instruments`.`size` != 'small';

Sequelize: queries in associations

- Multiple eager loading: the `include` option can receive an array in order to fetch multiple associated models at once

```
Foo.findAll({
  include: [
    {
      model: Bar,
      required: true
    },
    {
      model: Baz,
      where: /* ... */
    },
    /* ... */
  ]
})
```

Sequelize: queries in associations

- Eager loading N:M associations: when performing eager loading on a model with a Belongs-to-Many relationship, Sequelize fetchs the junction table data as well, by default

```
const Foo = sequelize.define('Foo', { name: DataTypes.TEXT });
const Bar = sequelize.define('Bar', { name: DataTypes.TEXT });
Foo.belongsToMany(Bar, { through: 'Foo_Bar' });
Bar.belongsToMany(Foo, { through: 'Foo_Bar' });

await sequelize.sync();
const foo = await Foo.create({ name: 'foo' });
const bar = await Bar.create({ name: 'bar' });
await foo.addBar(bar); // mixins
const fetchedFoo = await Foo.findOne({ include: Bar });
console.log(JSON.stringify(fetchedFoo, null, 2));
```

```
OUTPUT:
{
  "id": 1,
  "name": "foo",
  "Bars": [
    {
      "id": 1,
      "name": "bar",
      "Foo_Bar": {
        "FooId": 1,
        "BarId": 1
      }
    }
  ]
}
```

Sequelize: queries in associations

- Eager loading N:M associations: to remove the extra data from the junction table, one can explicitly provide an empty array to the `attributes` option inside the `through` option of the `include` option

```
const Foo = sequelize.define('Foo', { name: DataTypes.TEXT });
const Bar = sequelize.define('Bar', { name: DataTypes.TEXT });
Foo.belongsToMany(Bar, { through: 'Foo_Bar' });
Bar.belongsToMany(Foo, { through: 'Foo_Bar' });


await sequelize.sync();
const foo = await Foo.create({ name: 'foo' });
const bar = await Bar.create({ name: 'bar' });
await foo.addBar(bar); // mixins
const fetchedFoo = await Foo.findOne({
  include: {
    model: Bar,
    through: { attributes: [] }
  }
});
console.log(JSON.stringify(fetchedFoo, null, 2));
```

OUTPUT:

```
{
  "id": 1,
  "name": "foo",
  "Bars": [
    {
      "id": 1,
      "name": "bar",
    }
  ]
}
```

READ MORE ABOUT EAGER LOADING:
https://sequelize.org/docs/v6/advanced-association-concepts/eager-loading/

Sequelize: associations to fields not PK

- Sequelize allows to define an association that uses another field, instead of the primary key field, to establish the association
    - ➢ This other field must have a unique constraint on it (otherwise, it wouldn't make sense)
    - ➢ In the association options define the **target/source** key (depending on the relation type) and the foreign key name

```
const Ship = sequelize.define('ship', { name: DataTypes.TEXT });
const Captain = sequelize.define('captain', { name: { type: DataTypes.TEXT, unique: true }});

// This creates a foreign key called `captainName` in the source model (Ship)
// which references the `name` field from the target model (Captain)
Ship.belongsTo(Captain, { targetKey: 'name', foreignKey: 'captainName' });

await Captain.create({ name: "Jack Sparrow" });
const ship = await Ship.create({ name: "Black Pearl", captainName: "Jack Sparrow" });
console.log((await ship.getCaptain()).name); // "Jack Sparrow"
```

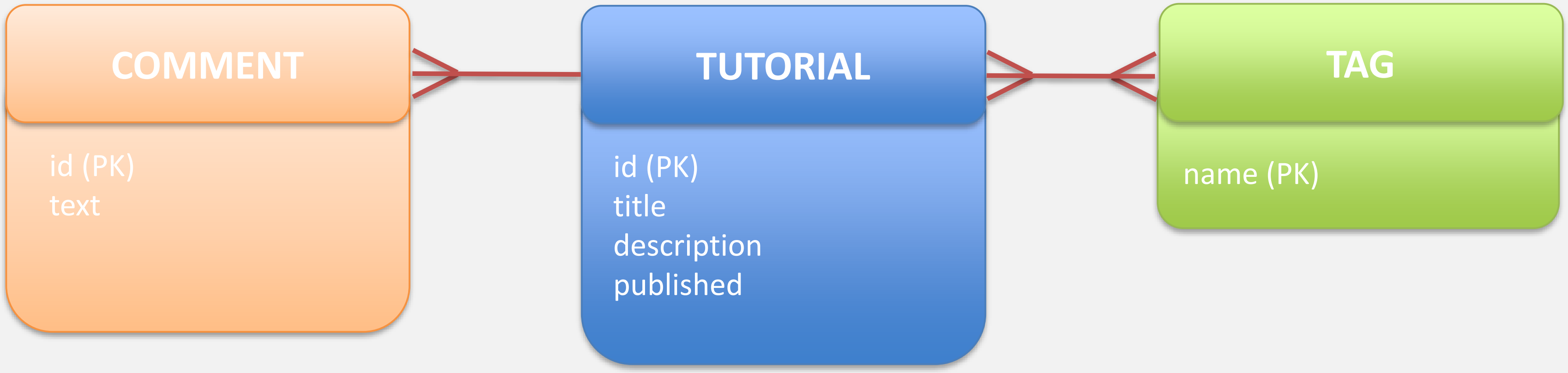
Sequelize: associations to fields not PK

- Sequelize allows to define an association that uses another field, instead of the primary key field, to establish the association
  - ➢ This other field must have a unique constraint on it (otherwise, it wouldn't make sense)
  - ➢ In the association options define the **target/source** key (depending on the relation type) and the foreign key name

```
const Foo = sequelize.define('foo', { name:  { type: DataTypes.TEXT, unique: true }});
const Bar = sequelize.define('bar', { title: { type: DataTypes.TEXT, unique: true }});

// This creates a junction table `foo_bar` with fields `fooName` and `barTitle`
Foo.belongsToMany(Bar, { through: 'foo_bar', sourceKey: 'name', targetKey: 'title' });
```

Relationships in Sequelize

- Let's continue our Tutorials REST API and present how to implement 1-N and N-N relationships and include some more routes to manipulate the new models
  - Assume that you want to design a Tutorial Blog data model, where one Tutorial can have no or many **Comments**, but one Comment only belongs to one Tutorial: **one-to-many** relationship
  - Assume also that one Tutorial has many **Tags**, and one Tag can point to many Tutorials: **many-to-many** relationship



Relationships in Sequelize

Start by adding the new models in the `models` folder

File *comments.model.js* on *models* folder

```
module.exports = (sequelize, DataTypes) => {
    const Comment = sequelize.define("comment", {
        text: {
            type: DataTypes.STRING,
            allowNull: false,
            validate: { notNull: { msg: "Text can not be empty!" } }
        }
    }, {
        timestamps: false
    });
    return Comment;
};
```

Relationships in Sequelize

Start by adding the new models in the `models` folder

File *tags.model.js* on *models* folder

```
module.exports = (sequelize, DataTypes) => {
    const Tag = sequelize.define("tag", {
        name: {
            type: DataTypes.STRING,
            primaryKey: true
        }
    }, {
        timestamps: false
    });
    return Tag;
};
```

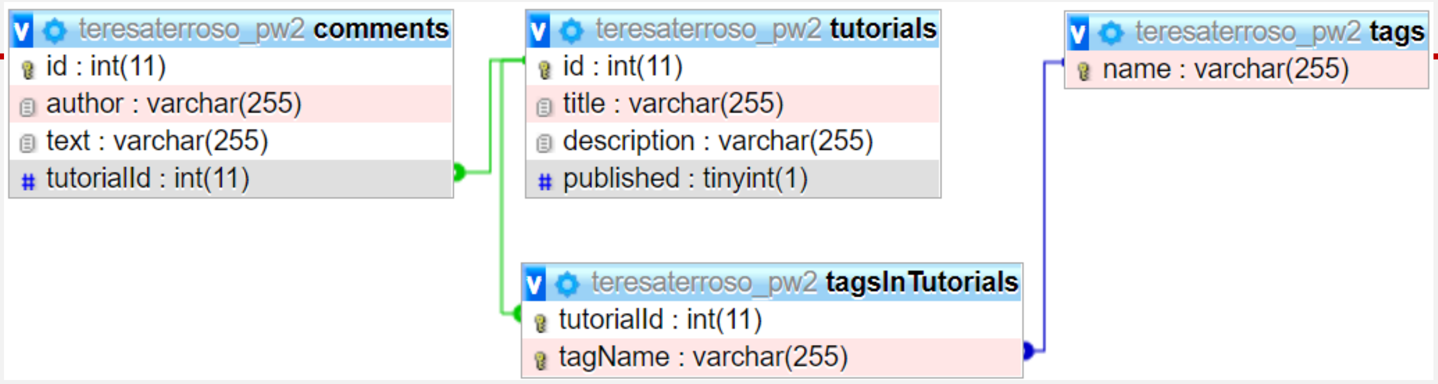
Relationships in Sequelize

Define the relationships between models

– synchronize the database if necessary

File *index.js* on *models* folder

```
…
//export the new models: COMMENT and TAG
db.comment = require("./comments.model.js")(sequelize, DataTypes);
db.tag = require("./tags.model.js")(sequelize, DataTypes);

//define the relationship 1:N between TUTORIAL and COMMENT models
db.tutorial.hasMany(db.comment); // if tutorial is deleted, delete all comments associated with it
db.comment.belongsTo(db.tutorial);

//define the relationship N:M between TUTORIAL and TAG models
db.tutorial.belongsToMany(db.tag, { through: 'tagsInTutorials', timestamps: false });
db.tag.belongsToMany(db.tutorial, { through: 'tagsInTutorials', timestamps: false });

…
```



Relationships in Sequelize

Model tutorial: add the following new routes to the API:

Verb	URI	Description
POST	tutorials/{idT}/comments	Adds a new comment to a given tutorial
GET	tutorials/{idT}/comments	Get all comments of a given tutorial
DELETE	tutorials/{idT}/comments/{idC}	Deletes a given comment from a given tutorial
PATCH	tutorials/{idT}/comments/{idC}	Allows to alter (only) the text of a given comment from a given tutorial

Relationships in Sequelize

Model tag: add the following new routes to the API:

Verb	URI	Description
POST	tags	Creates a new tag
GET	tags	List all tags
PUT	tutorials/{idT}/tags/{idTag}	Adds a given tag into a given tutorial
DELETE	tutorials/{idT}/tags/{idTag}	Deletes a given tag from a given tutorial

Also, alter some of the 'old' routes to the API:

Verb	URI	Description
GET	tutorials/{idT}	Include the comments and tags for a given tutorial

Relationships in Sequelize

- All new routes (except one) start with '`/tutorials/{idT}`' but all refer to a new model, `comments`
- These routes can be added to the project in two different ways:
  1. Add all the new routes in the already existing `tutorials.routes.js` file of `routes` folder
  2. **Or** create a new file `comments.routes.js` file into `routes` folder, and add those routes as **middleware of the tutorials routes** (hierarchical or nested routes)

```
…
const commentsRouter =
require("./comments.routes");
…
// you can nest routers by attaching them as
middleware
router.use('/:tutorialID/comments',
commentsRouter);
```

`tutorials.routes.js`

```
…
const commentController =
require("../controllers/comments.controller");

// set 'mergeParams: true' on the router to access params
from the parent router (like tutorialID req parameter)
let router = express.Router({mergeParams: true});
…
router.route('/')
     .post(commentController.createComment);
```

`comments.routes.js`

Relationships in Sequelize

Complete the controllers files to implement the new routes

– Example of the function to process a POST request to /tutorials/{idT}/comments

File *comments.controller.js* on *controllers* folder

```
const db = require("../models/index.js");
const Tutorial = db.tutorial; const Comment = db.comment;

exports.create = async (req, res) => {
    try {
        // try to find the tutorial, given its ID
        let tutorial = await Tutorial.findByPk(req.params.idT)
        if (tutorial === null)
            return res.status(404).json({
                success: false, msg: `Cannot find any tutorial with ID ${req.params.idT}.`
            });

        // save Comment in the database
        let newComment = await Comment.create(req.body);
        // add Comment to found tutorial (using a mixin)
        await tutorial.addComment(newComment);

        res.status(201).json({
                success: true, msg: `Comment added to tutorial with ID ${req.params.idT}.`,
                URL: `/tutorials/${req.params.idT}/comments/${newComment.id}` });
    }
```

```
//SAME AS
let newComment = await Comment.create({
   author: req.body.author,
   text: req.body.text,
   tutorialId: req.params.idT
});
```

Relationships in Sequelize

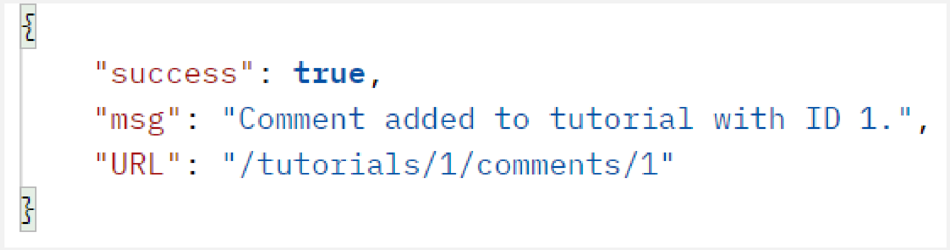
Complete the controllers files to implement the new routes.

– Example of the route `POST /tutorials/{idT}/comments`

File *comments.controller.js* on *controllers* folder

```
...
    catch (err) {
        if (err instanceof ValidationError)
            res.status(400).json({ success: false, msg: err.errors.map(e => e.message) });
        else
            res.status(500).json({
                success: false,
                msg: err.message
                    || `Some error occurred while adding a comment to tutorial with ID ${req.params.idT}.`
            });
    };
};
```

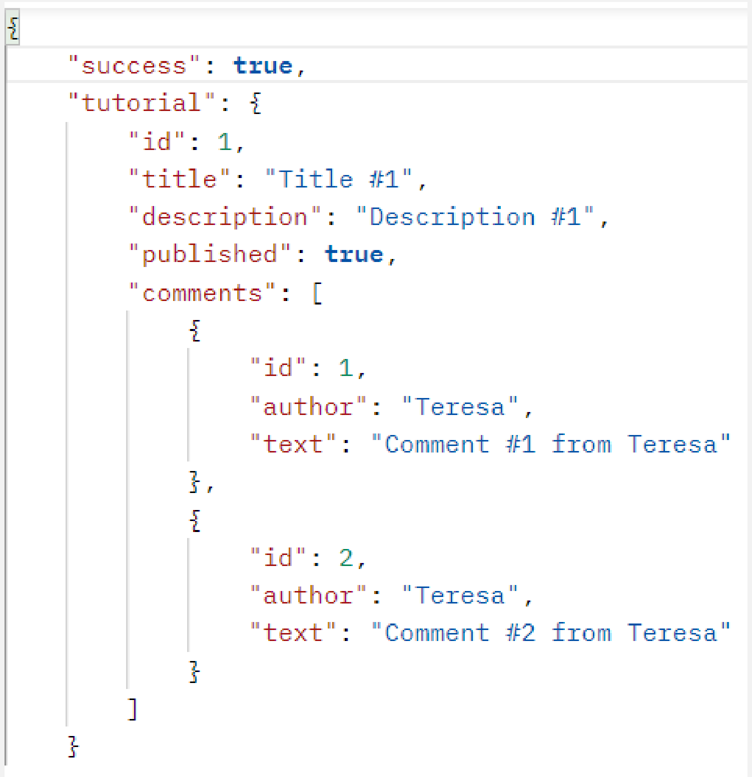
SUCCESSFUL RESPONSE example



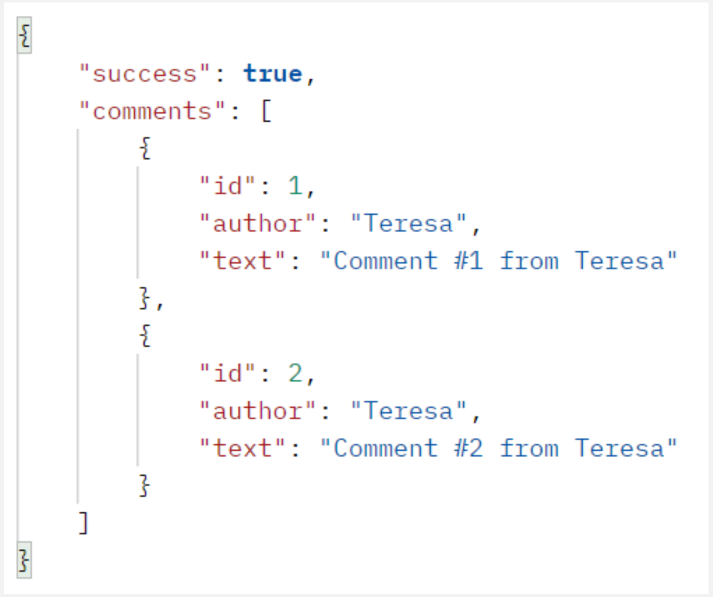
```
{
    "success": true,
    "msg": "Comment added to tutorial with ID 1.",
    "URL": "/tutorials/1/comments/1"
}
```

Relationships in Sequelize

- Response example of route `GET tutorials/{id}/comments`:
  - Try to obtain two different response formats: with and without the tutorial data
  - In both cases, remove the `tutorialId` info from the comments, since it is implicit in the request URL



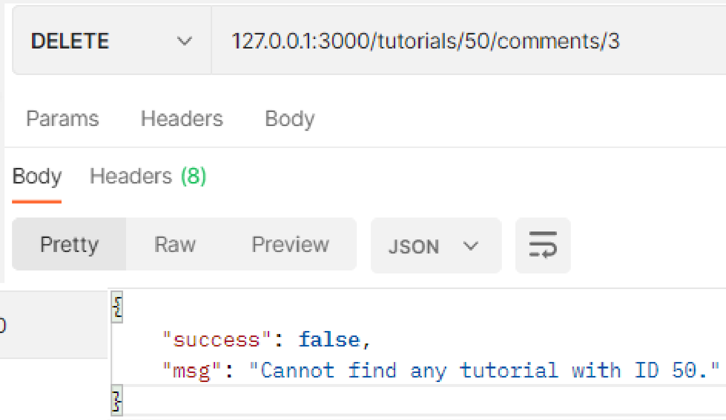
```
{
    "success": true,
    "tutorial": {
        "id": 1,
        "title": "Title #1",
        "description": "Description #1",
        "published": true,
        "comments": [
            {
                "id": 1,
                "author": "Teresa",
                "text": "Comment #1 from Teresa"
            },
            {
                "id": 2,
                "author": "Teresa",
                "text": "Comment #2 from Teresa"
            }
        ]
    }
```



```
{
    "success": true,
    "comments": [
        {
            "id": 1,
            "author": "Teresa",
            "text": "Comment #1 from Teresa"
        },
        {
            "id": 2,
            "author": "Teresa",
            "text": "Comment #2 from Teresa"
        }
    ]
}
```

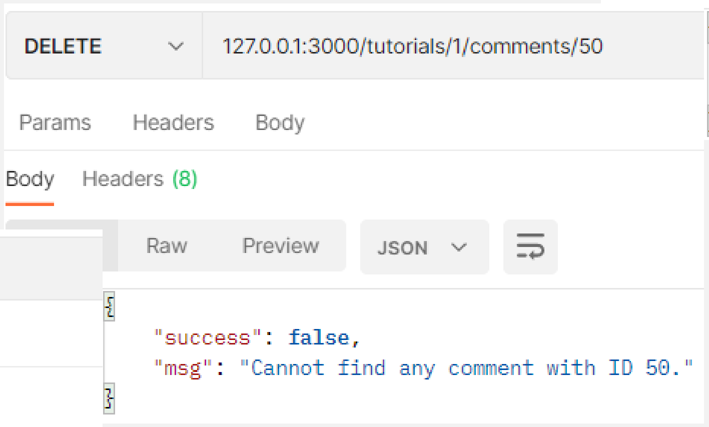
Relationships in Sequelize

- Routes `PATCH tutorials/{idT}/comments/{idC}` and `DELETE tutorials/{idT}/comments/{idC}`:
  - make sure to check if:
    1. Tutorial exists
    2. Comment exists
    3. Comment belong to tutorial



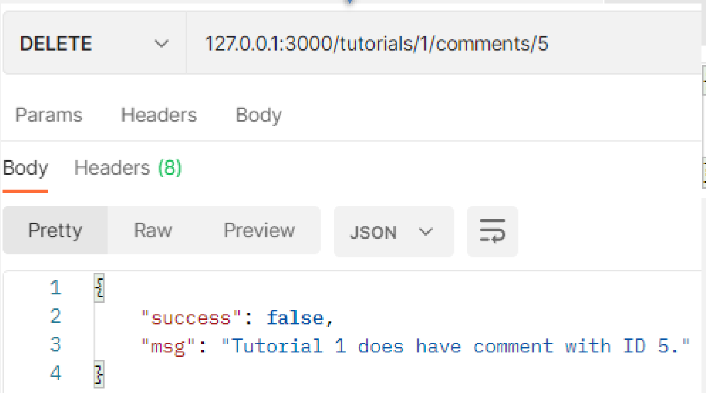
DELETE 127.0.0.1:3000/tutorials/50/comments/3

```
{
    "success": false,
    "msg": "Cannot find any tutorial with ID 50."
}
```



DELETE 127.0.0.1:3000/tutorials/1/comments/50

```
{
    "success": false,
    "msg": "Cannot find any comment with ID 50."
}
```

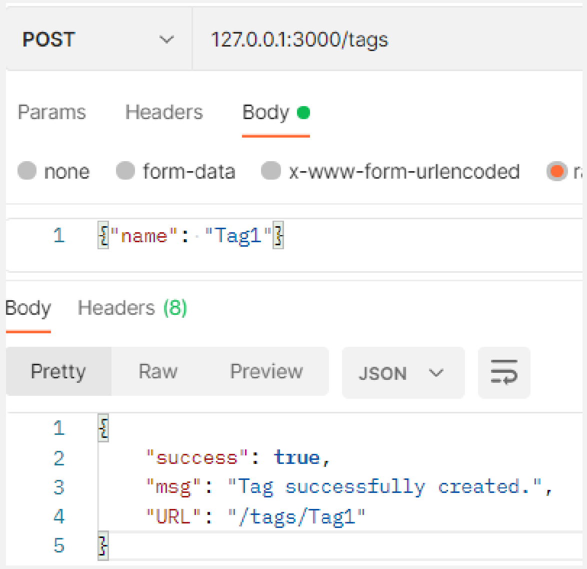


DELETE 127.0.0.1:3000/tutorials/1/comments/5

```
{
    "success": false,
    "msg": "Tutorial 1 does have comment with ID 5."
}
```

Relationships in Sequelize

- Route `POST tags`:
  - make sure to check for tag name uniqueness:



POST 127.0.0.1:3000/tags

Params Headers Body

none form-data x-www-form-urlencoded

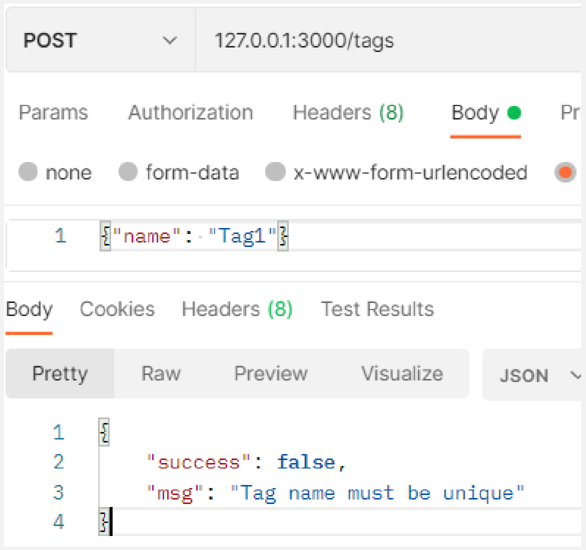
```
{"name": "Tag1"}
```

Body Headers (8)

Pretty Raw Preview JSON

```
{
    "success": true,
    "msg": "Tag successfully created.",
    "URL": "/tags/Tag1"
}
```

1st call



POST 127.0.0.1:3000/tags

Params Authorization Headers (8) Body Pr

none form-data x-www-form-urlencoded

```
{"name": "Tag1"}
```

Body Cookies Headers (8) Test Results

Pretty Raw Preview Visualize JSON

```
{
    "success": false,
    "msg": "Tag name must be unique"
}
```

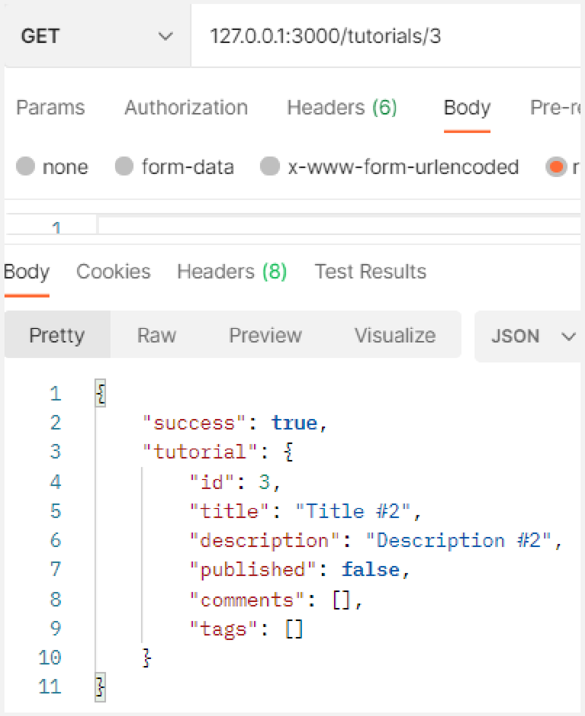
2nd call (with the same body parameters)

Relationships in Sequelize

- Routes `PUT tutorials/{idT}/tags/{idTag}` and `DELETE tutorials/{idT}/tags/{idTag}`:
  - make sure to check if:
  1. Tutorial exists
  2. Tag exists
  3. On adding a new tag, make sure the tutorial does not already have it / when deleting a tag, make sure the tutorial has that tag

Relationships in Sequelize

- Route `GET tutorial/{idT}`:
  - include the tutorial comments and tags (if any):



GET 127.0.0.1:3000/tutorials/3

Params Authorization Headers (6) Body Pre-r

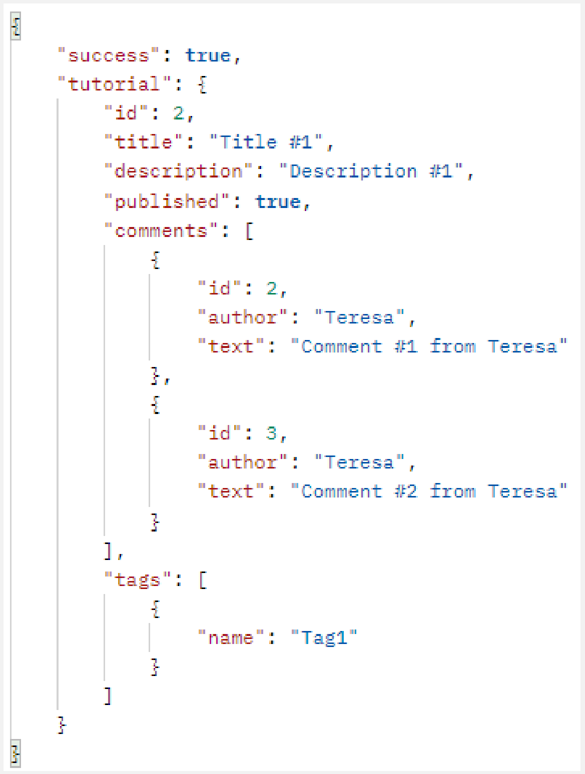
none form-data x-www-form-urlencoded r

Body Cookies Headers (8) Test Results

Pretty Raw Preview Visualize JSON

```
{
    "success": true,
    "tutorial": {
        "id": 3,
        "title": "Title #2",
        "description": "Description #2",
        "published": false,
        "comments": [],
        "tags": []
    }
}
```

Tutorial without tags or comments

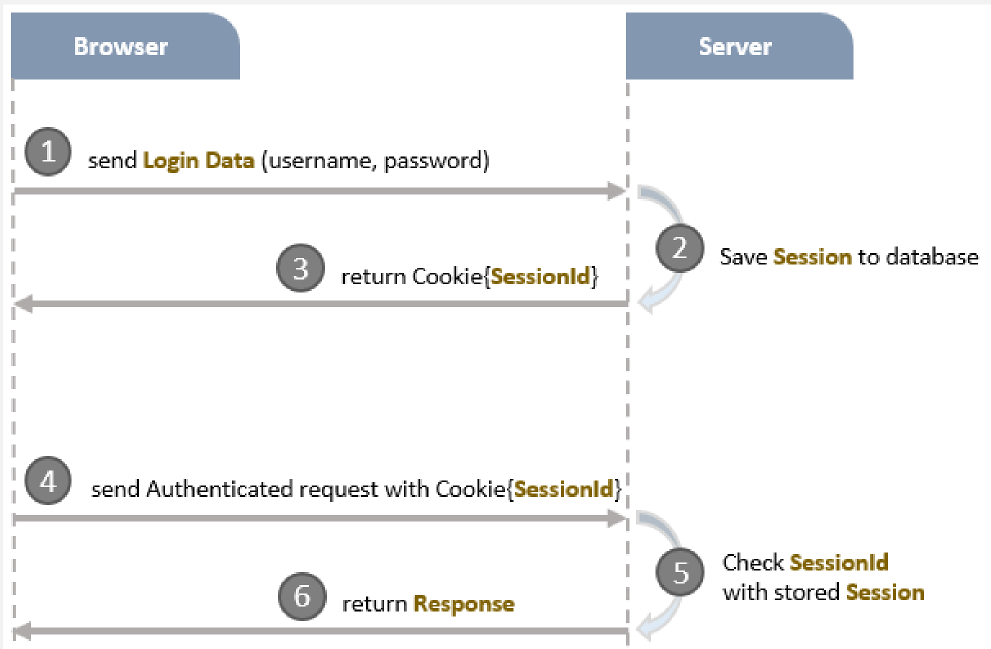


```
{
    "success": true,
    "tutorial": {
        "id": 2,
        "title": "Title #1",
        "description": "Description #1",
        "published": true,
        "comments": [
            {
                "id": 2,
                "author": "Teresa",
                "text": "Comment #1 from Teresa"
            },
            {
                "id": 3,
                "author": "Teresa",
                "text": "Comment #2 from Teresa"
            }
        ],
        "tags": [
            {
                "name": "Tag1"
            }
        ]
    }
}
```

Tutorial with 2 comments and one tag

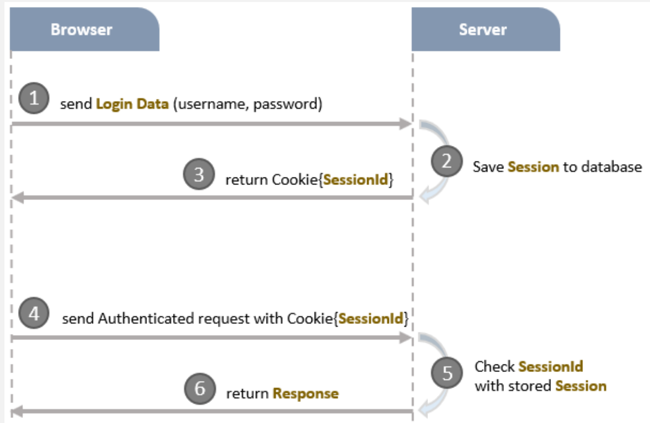
Authentication

- Authentication is one of the most important parts in almost applications
  - ➢ For using any website, mobile app or desktop app you may need to create an account, then use it to login for accessing features of the app: **authentication**
- So, how to authenticate an account?
  Simple method that popular websites used in the past:
  Session-based Authentication


Browser
Server
1 send Login Data (username, password)
2 Save Session to database
3 return Cookie{SessionId}
4 send Authenticated request with Cookie{SessionId}
5 Check SessionId with stored Session
6 return Response

Authentication

- **Session-based Authentication**:
  1. when a user logs into a website, the server will generate a **Session** for that user and store it (in memory or database)
  2. server also returns a **SessionId** for the client to save it in browser **cookie**
  3. the Session on Server has an expiration time; after that time, this session expires, and the user must re-login to create another session
  4. if the user has logged in and the session has not expired yet, the cookie (including **SessionId**) always goes with all HTTP Request to Server
  5. server will compare this **SessionId** with stored session to authenticate and return corresponding response

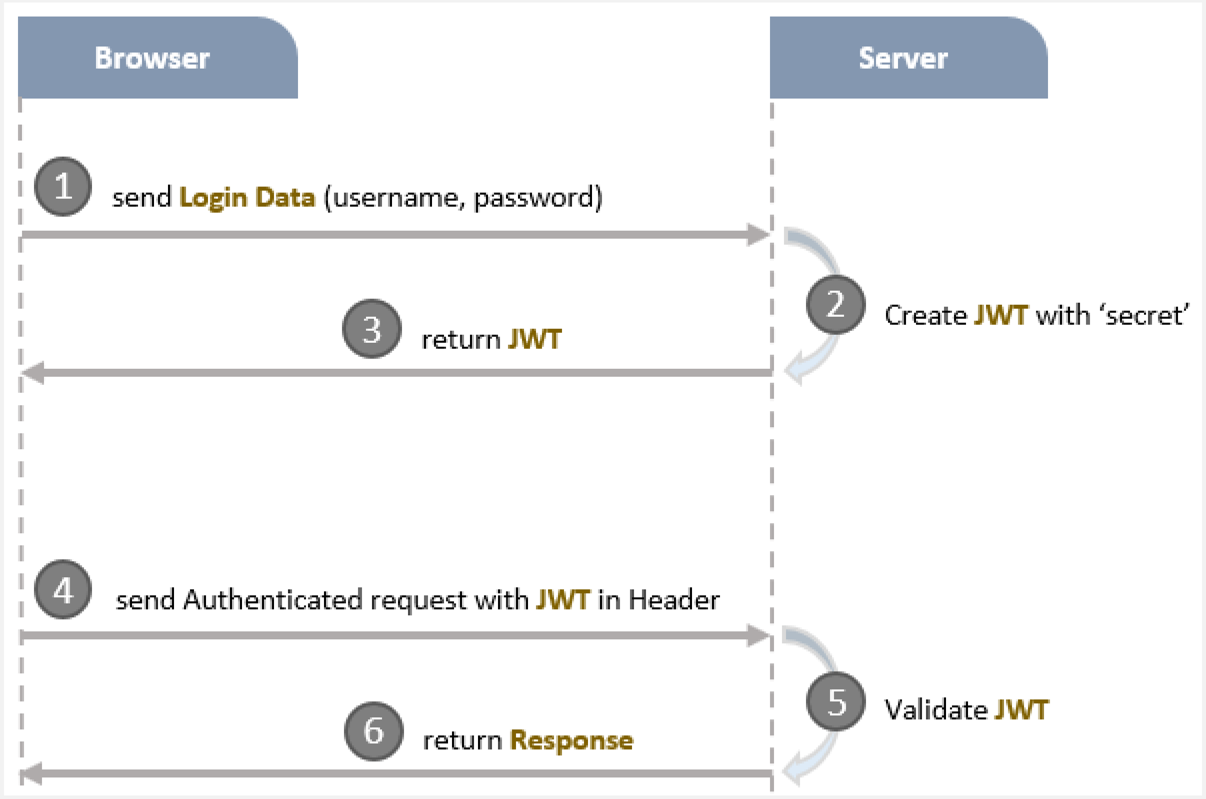

Browser
Server
1 send Login Data (username, password)
2 Save Session to database
3 return Cookie{SessionId}
4 send Authenticated request with Cookie{SessionId}
5 Check SessionId with stored Session
6 return Response

Authentication

- Let's imagine one day, you want to implement system for mobile (Native Apps) and use the same database of the current web app. You cannot authenticate users who use native app using Session-based Authentication because mobile apps <u>don’t use cookies</u> in the same way as web applications
    - ➢ Should one build another backend project that supports native apps?
    - ➢ Or should one write an authentication module for native app users?
- That’s why **Token-based Authentication** was born
    - ➢ Nowadays many RESTful APIs use it

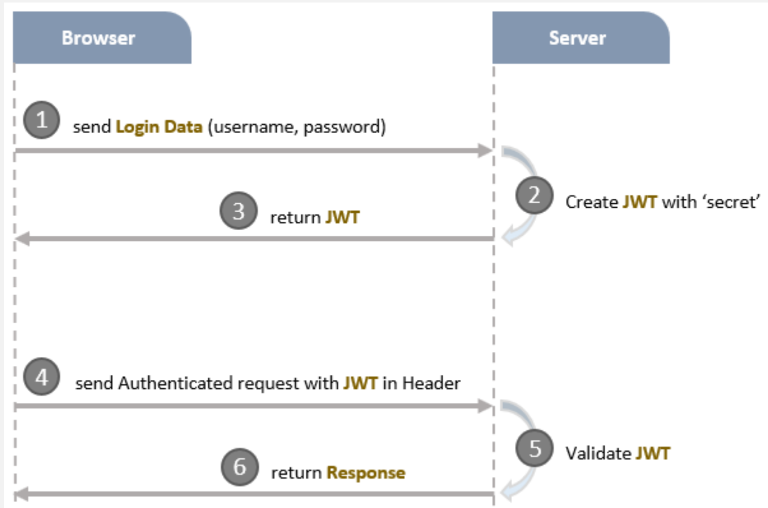
Authentication

- How **Token-based Authentication** works:


Browser
Server
1 send Login Data (username, password)
2 Create JWT with 'secret'
3 return JWT
4 send Authenticated request with JWT in Header
5 Validate JWT
6 return Response

Authentication

- How **Token-based Authentication** works:
  1. Instead of creating a session, the server generates a JWT (**JSON Web Token**) from user login data and send it to the client
  2. The client saves the JWT and from that point, every request from client that requires authentication should be attached that JWT (in authorization header - with or without a **Bearer** prefix - or in x-access-token header)
  3. The server validates the JWT and returns the response
- For storing JWT on Client side, it depends on the platform:
  - ➢ Browser: [local storage]
  - ➢ IOS: [Keychain]
  - ➢ Android: [SharedPreferences]

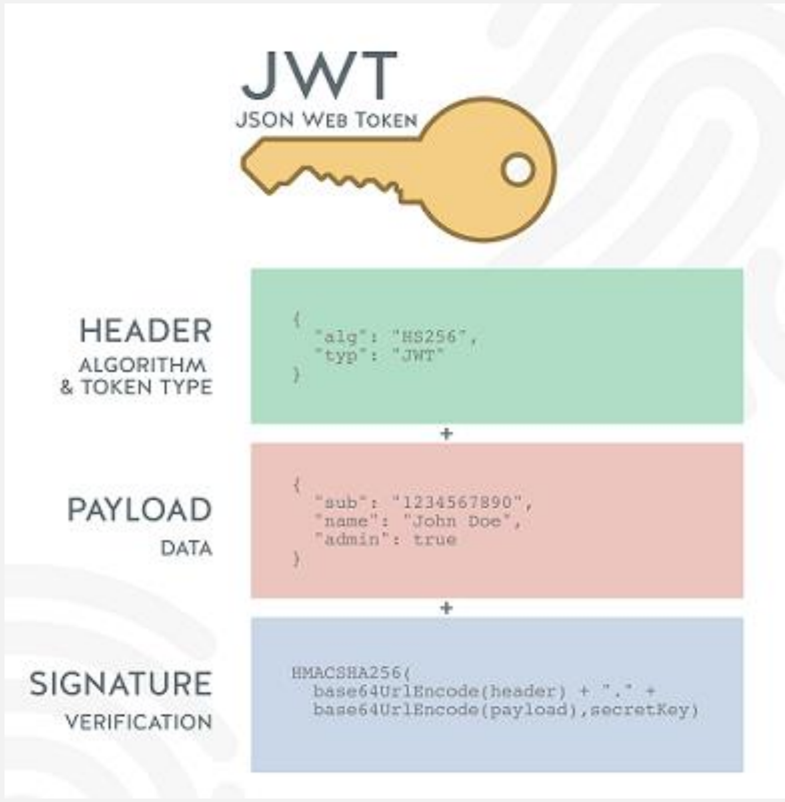

Browser
Server
1 send Login Data (username, password)
2 Create JWT with 'secret'
3 return JWT
4 send Authenticated request with JWT in Header
5 Validate JWT
6 return Response

JWT – JSON Web Token

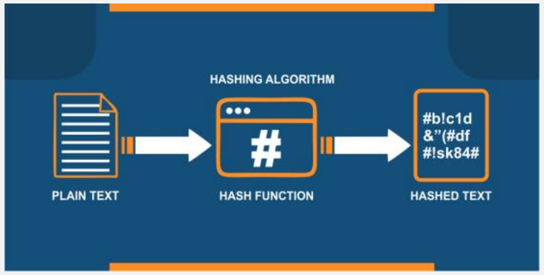
- JSON Web Token (JWT) is an open standard (RFC 7519) that defines a **compact** and self-contained way for **securely transmitting information** between parties as a **JSON object**
- The transmitted information can be **verified** and **trusted** because it is digitally signed
- JWTs can be signed using a secret or a public/private key pair
- Authorization Headers: `Authorization: <type> <credentials>`
  - Pattern introduced by the W3C in HTTP 1.0
  - Apps that use this pattern are more than likely implementing OAuth 2.0 bearer tokens

JWT – JSON Web Token

- Important parts of a JWT:
  1. Header
  2. Payload
  3. Signature
- Header tells how to calculate JWT: `typ` means type and indicates the Token type (here is JWT) `alg` stands for algorithm, which is a [hash algorithm]() for generating token signature

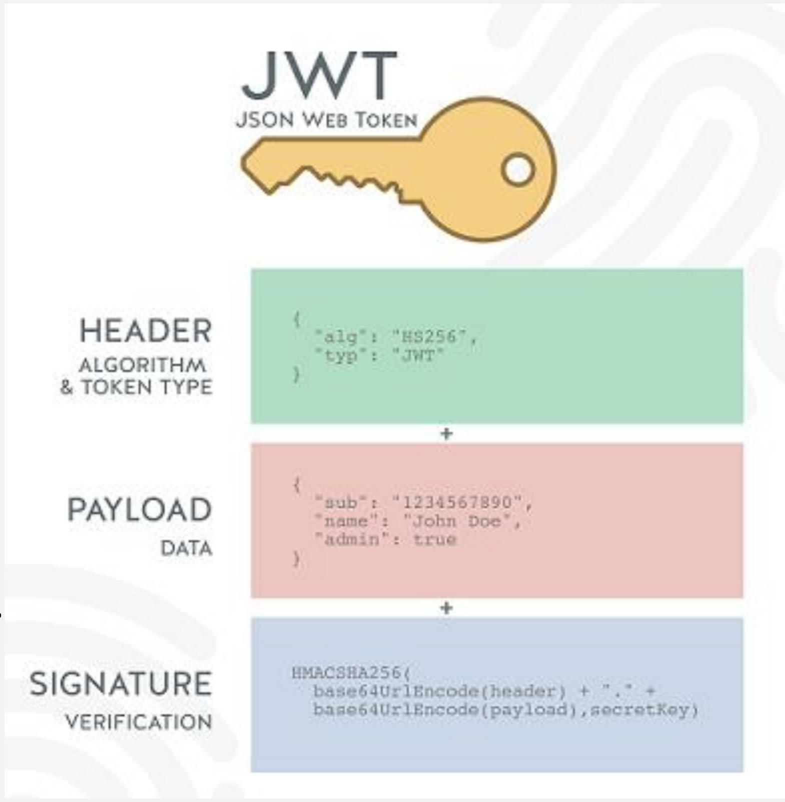


```
{
  "alg": "HS256",
  "typ": "JWT"
}
```
+
```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```
+
```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),secretKey)
```



JWT – JSON Web Token

- Important parts of a JWT:
  1. Header
  2. Payload
  3. Signature
- Payload tells what to store in JWT: standard fields are
  `iss` issuer - who issues the JWT
  `iat` stands for 'issued at', time the JWT was issued at
  `exp` JWT expiration time
  … other user fields



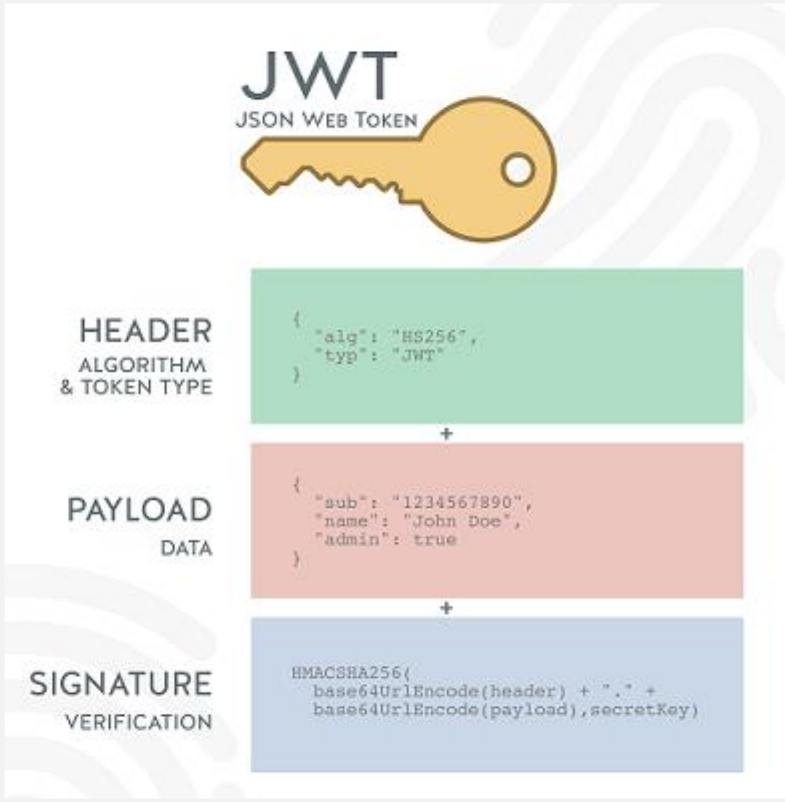
```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),secretKey)
```

JWT – JSON Web Token

- Important parts of a JWT:
  1. Header
  2. Payload
  3. Signature
- Signature encodes the header and payload, and concatenates the 2 together with a period separator; that string is then run through the cryptographic algorithm (hash algorithm specified in the header) with a secret string



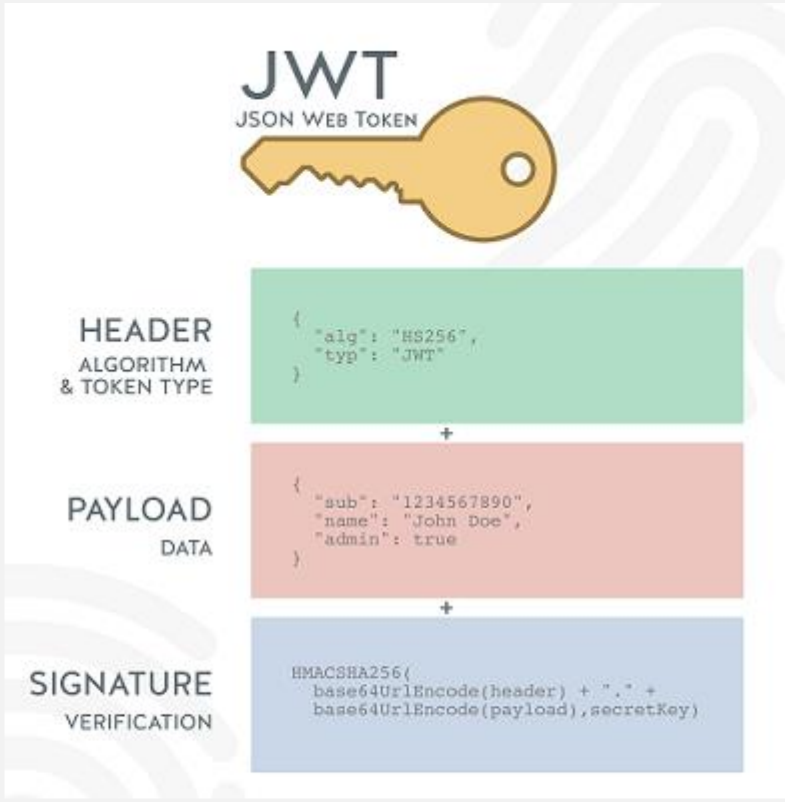
```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),secretKey)
```

JWT – JSON Web Token

- Important parts of a JWT:
  1. Header
  2. Payload
  3. Signature
- The 3 parts are encoded separately and concatenated using periods to produce the JWT token, easily passed using HTTP

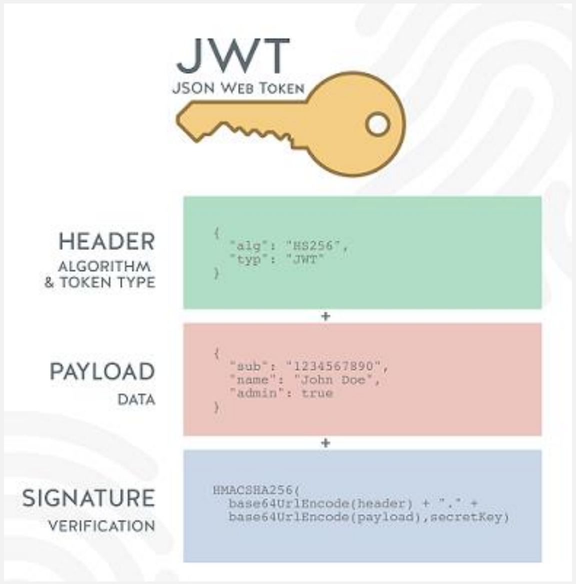


```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),secretKey)
```

JWT – JSON Web Token



```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),secretKey)
```

- Advantages:
  - ➢ This is a **stateless authentication** mechanism as the user state is never saved in server memory
  - ➢ The server's **protected routes** will check for a valid JWT in the authorization header, and if it is present, the user will be allowed to access them
  - ➢ As JWTs are **self-contained**, all the necessary information is there, reducing the need to query the database multiple times
- Beware that JWT does NOT secure your data
  - ➢ The process of generating JWT (Header, Payload, Signature) only encode & hash data, not **encrypt** data
  - ➢ The purpose of JWT is to prove that the data is generated by an authentic source
  - ➢ So, what if there is a Man-in-the-middle attack that can get JWT, then decode user information? Yes, that is possible, so always make sure that your application has the HTTPS encryption

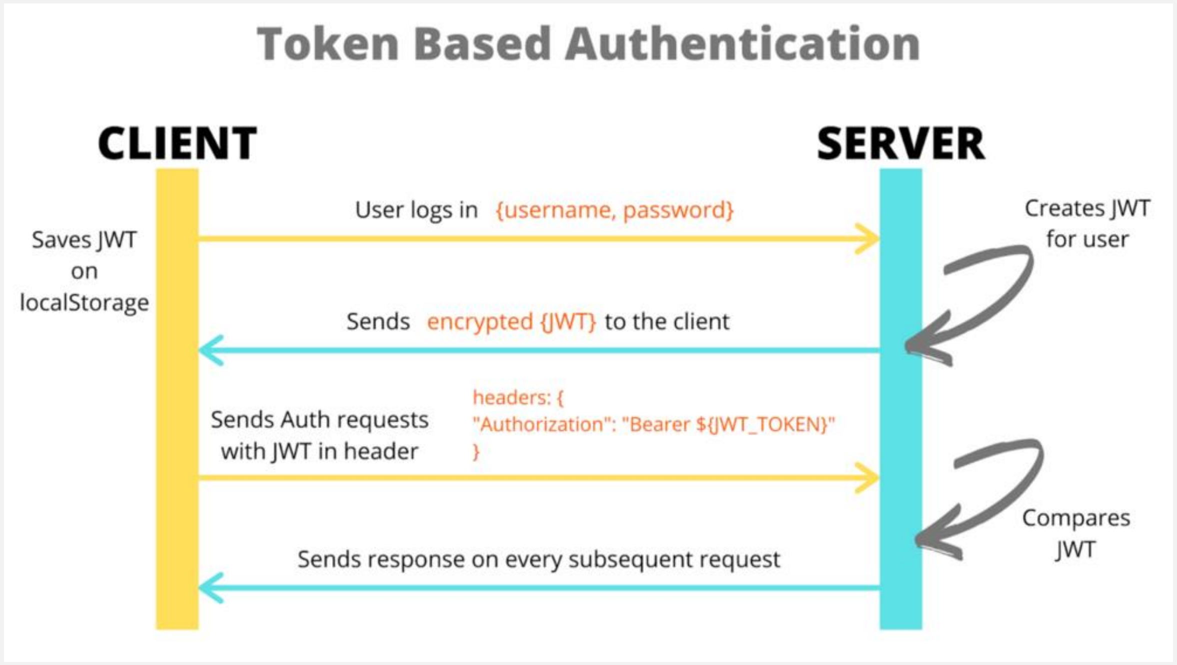
JWT – JSON Web Token

- How Server validates JWT from client:
  - Server uses a **secret string to create signature**; this secret is unique for every application and must be **stored securely** in the server side
  - When receiving JWT from client, the server gets the signature, verifies that the signature is correctly hashed by the same algorithm and secret string; if it matches the server's signature, the JWT is valid
- Further reading: RFC7519 – JSON Web Token (JWT)

JWT Signup & Login example

- Node Modules:
  jsonwebtoken – implementation of JSON Web Tokens
  bcryptjs – implements hash algorithms
- Install dependencies:
  `npm install jsonwebtoken bcryptjs --save`
- Let's build a Node.js Express application that:
  - Has a model **User**, with username (unique), email (unique), password and role ('admin' or 'regular')
  - User can signup for a new account, or login with username & password
  - The API authorizes the user to access protected resources with JWT tokens and some routes check for authorization levels like ID or role

JWT Signup & Login example


Token Based Authentication
CLIENT
SERVER
User logs in {username, password}
Creates JWT for user
Saves JWT on localStorage
Sends encrypted {JWT} to the client
Sends Auth requests with JWT in header
headers: {
"Authorization": "Bearer ${JWT_TOKEN}"
}
Compares JWT
Sends response on every subsequent request

Flow of User Registration, User Login and Authorization process

A legal JWT must be added to HTTP Authorization Header if Client accesses protected resources

JWT Signup & Login example

- Set up the usual directory structure for the Node.js Express application: config, controllers, routes and models folders
- Add the API secret string into the environment variables and read it into the file `config.js` at config folder

```
...
# secret key to encode and decode JWT token
SECRET: "my-API-ultra-secure-and-ultra-long-secret"
```

`.env`

```
module.exports = {
      ...
      SECRET: process.env.SECRET
};
```

`config.js`

`jsonwebtoken` module functions such as **`verify()`** or **`sign()`** use a hash algorithm that needs a secret key (as String) to encode and decode token

JWT Signup & Login example

- In the models folder (and assuming the usage of a MySQL database), add the following model for users

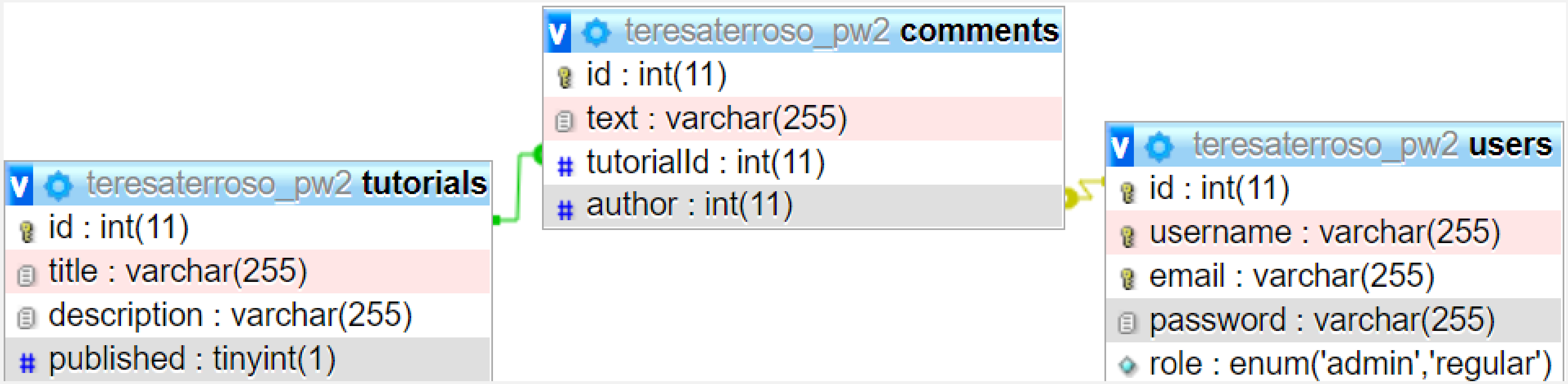
```
module.exports = (sequelize, DataTypes) => {
    const User = sequelize.define("user", {
        username: {
            type: DataTypes.STRING,
            unique: true,
            allowNull: false,
            validate: {
notNull: { msg: "Username cannot be empty or null!" } }
        },
        email: {
            type: DataTypes.STRING
        },
        password: {
            type: DataTypes.STRING,
            trim: true, // remove spaces on both
ends                 allowNull: false,
            validate: {
notNull: { msg: "Password cannot be empty or null!" } }
        },
       ...
```

File *user.model.js* on *models* folder

```
...
        role: {
            type: DataTypes.ENUM('admin', 'regular'),
            defaultValue: 'regular',
            validate: {isIn: {
                args: [['admin', 'regular']],
                msg: "Allowed roles: admin or regular"
              }}
        }
    }, { timestamps: false });
    return User;
};
```

JWT Signup & Login example

- In the models folder, maintain the tutorials and comments models, however **remove the author field** from the comment model, since this will be a **foreign key**, relating comments with users



- Define the 1:N relationship between `users` and `comments`

```
// if user is deleted, do not delete comments associated with it (set author as null) - default Sequelize behaviour on 1:N relationships
db.user.hasMany(db.comment, { foreignKey: 'author' });
db.comment.belongsTo(db.user, { foreignKey: 'author' });
```

File *index.js* on *models* folder

JWT Signup & Login example

- Add the following routes for request URL's starting with **`/users`**:

File *users.routes.js* on *routes* folder

```
const express = require('express');
const authController = require("../controllers/auth.controller");
const userController = require("../controllers/users.controller");

let router = express.Router();


router.route('/')
    .get(authController.verifyToken, userController.getAllUsers) //ADMIN ACCESS ONLY
    .post(userController.create); //PUBLIC

router.route('/login')
    .post(userController.login); //PUBLIC

router.route('/:userID')
     .get(authController.verifyToken, userController.getUser); //ADMIN or LOGGED USER ONLY

router.all('*', function (req, res) {
    res.status(404).json({ message: USERS: what???' });
})

module.exports = router;
```

Authentication middleware for protected routes

JWT Signup & Login example

- Add the corresponding route handlers in the controllers folder:

File *user.controller.js* on *controllers* folder

```
const jwt = require("jsonwebtoken"); //JWT tokens creation (sign())
const bcrypt = require("bcryptjs");  //password encryption

const config = require("../config/config.js");
const db = require("../models");
const User = db.user;

const { ValidationError } = require('sequelize');

exports.create = async (req, res) => {...};

exports.login = async (req, res) => {...};

exports.getAllUsers = async (req, res) => {...};

exports.getUser = async (req, res) => {...};
```

JWT Signup & Login example

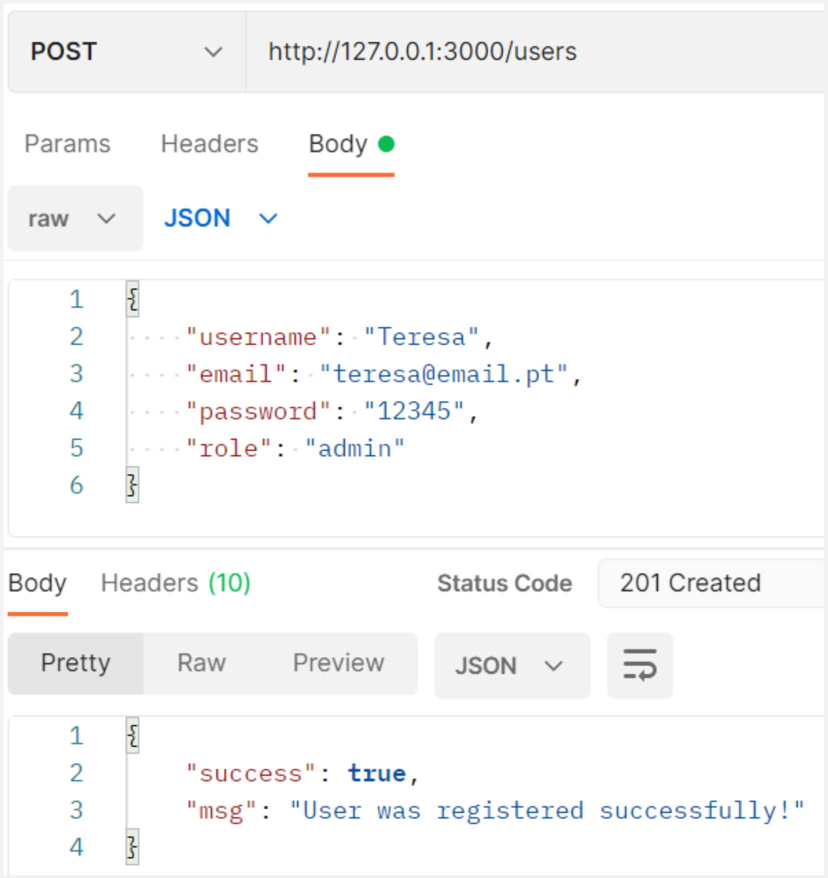
- Example for user creation (signup): do not save user's passwords without being **hashed***

HASH: Unlike encryption which you can decode to get back the original password, hashing is a **one-way** function that can't be reversed once done

File *user.controller.js* on *controllers* folder

```
exports.create = async (req, res) => {
    try {
        if (!req.body && !req.body.username && !req.body.password)
            return res.status(400).json({ success: false, msg: "Username and password are mandatory" });

        // Save user to DB
        await User.create({
            username: req.body.username, email: req.body.email,
            // hash its password (8 = #rounds – more rounds, more time)
            password: bcrypt.hashSync(req.body.password, 10),
            role: req.body.role
        });

        return res.status(201).json({ success: true, msg: "User was registered successfully!" });
    }
    catch (err) {
        if (err instanceof ValidationError)
            res.status(400).json({success: false, msg: err.errors.map(e => e.message)});
        els
            res.status(500).json({success: false, msg: err.message||"Some error occurred while signing up."});
  };
};
```

JWT Signup & Login example

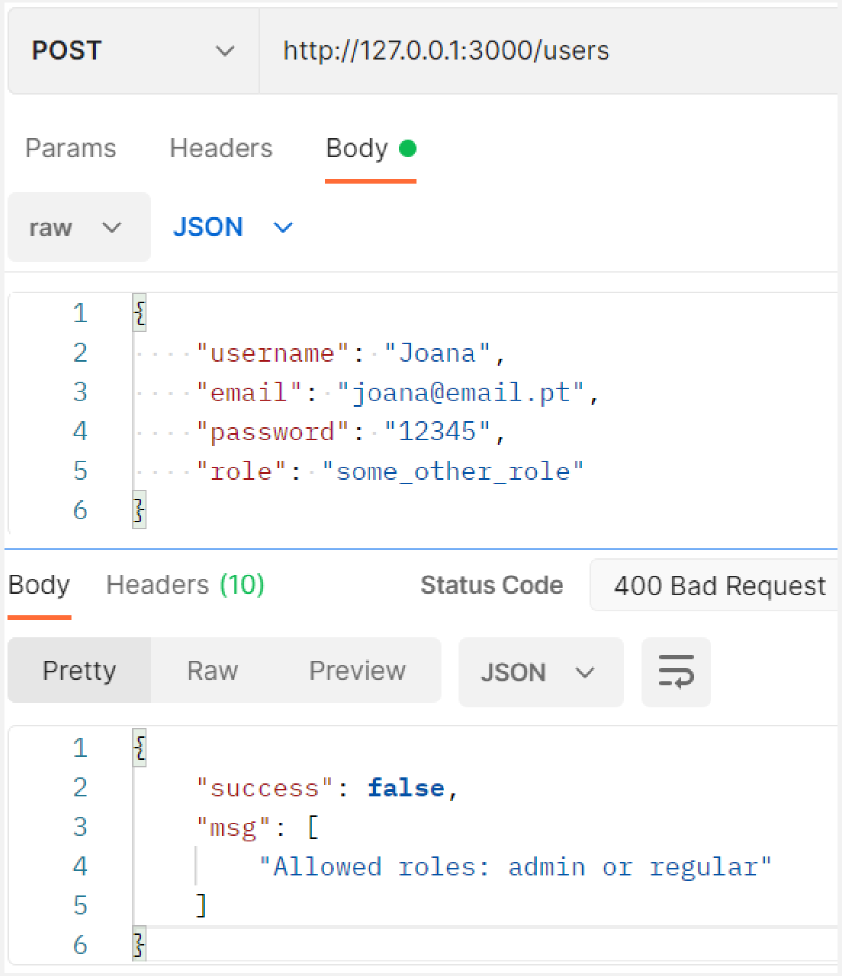


http://127.0.0.1:3000/users

```
{
    "username": "Teresa",
    "email": "teresa@email.pt",
    "password": "12345",
    "role": "admin"
}
```

201 Created

```
{
    "success": true,
    "msg": "User was registered successfully!"
}
```



http://127.0.0.1:3000/users

```
{
    "username": "Joana",
    "email": "joana@email.pt",
    "password": "12345",
    "role": "some_other_role"
}
```

400 Bad Request

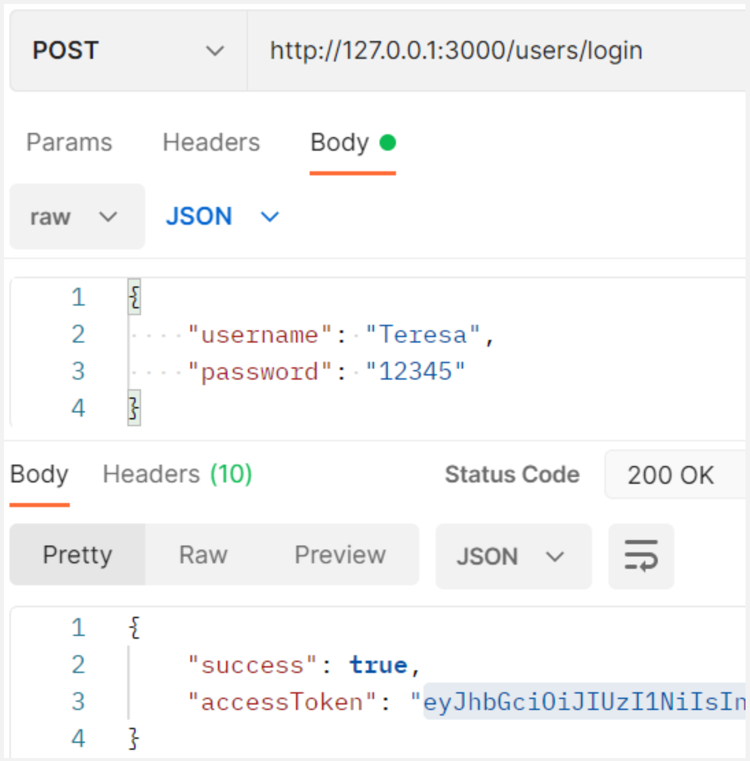
```
{
    "success": false,
    "msg": [
        "Allowed roles: admin or regular"
    ]
}
```

JWT Signup & Login example

- Example for user login: verify user's password and create JWT

```
exports.login = async (req, res) => {
    try {
        if (!req.body || !req.body.username || !req.body.password)
            return res.status(400).json({ success: false, msg: "Must provide username and password." });

        let user = await User.findOne({ where: { username: req.body.username } }); //get user data from DB
        if (!user) return res.status(404).json({ success: false, msg: "User not found." });

        // tests a string (password in body) against a hash (password in database)
        const check = bcrypt.compareSync( req.body.password, user.password );
        if (!check) return res.status(401).json({ success:false, accessToken:null, msg:"Invalid credentials!" });

        // sign the given payload (user ID and role) into a JWT payload – builds JWT token, using secret key
        const token = jwt.sign({ id: user.id, role: user.role },
            config.SECRET, { expiresIn: '24h' // 24 hours
        });
        return res.status(200).json({ success: true, accessToken: token });

    } catch (err) {
        if (err instanceof ValidationError)
            res.status(400).json({ success: false, msg: err.errors.map(e => e.message) });
        else
            res.status(500).json({ success: false, msg: err.message || "Some error occurred at login."});
    };
};
```

JWT Signup & Login example



POST http://127.0.0.1:3000/users/login

Params Headers Body ●

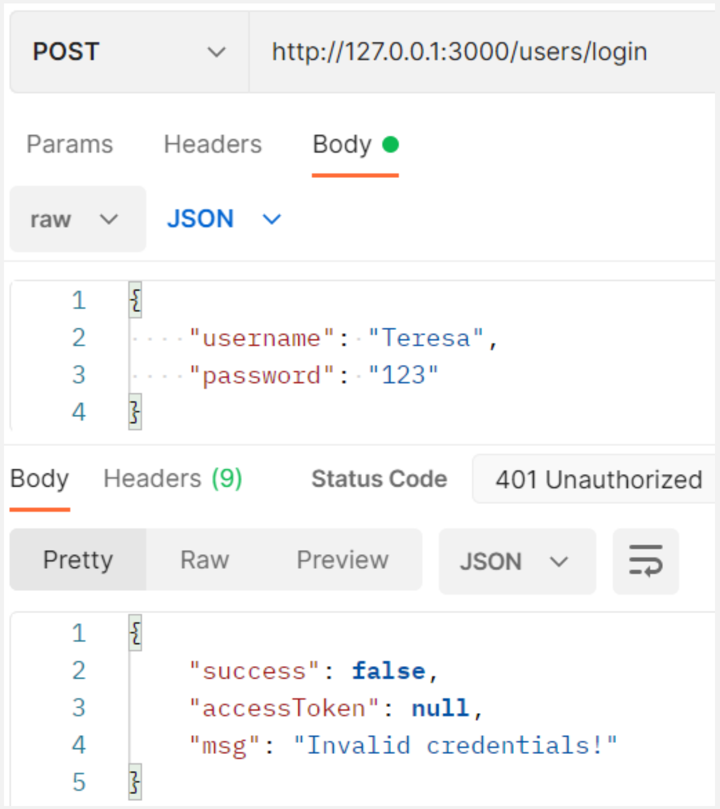
raw JSON

```
{
    "username": "Teresa",
    "password": "12345"
}
```

Body Headers (10) Status Code 200 OK

Pretty Raw Preview JSON

```
{
    "success": true,
    "accessToken": "eyJhbGciOiJIUzI1NiIsIn
}
```



POST http://127.0.0.1:3000/users/login

Params Headers Body ●

raw JSON

```
{
    "username": "Teresa",
    "password": "123"
}
```

Body Headers (9) Status Code 401 Unauthorized

Pretty Raw Preview JSON

```
{
    "success": false,
    "accessToken": null,
    "msg": "Invalid credentials!"
}
```

JWT Signup & Login example

- Add the authentication middleware function in `auth.controller.js`:

```
…
exports.verifyToken = (req, res, next) => {
    // search token in headers most commonly used for authorization
    const header = req.headers['x-access-token'] || req.headers.authorization;
    if (typeof header == 'undefined')
        return res.status(401).json({ success: false, msg: "No token provided!"  });

    const bearer = header.split(' '); // Authorization header format: Bearer <token>
    const token = bearer[1];

    try {
        let decoded = jwt.verify(token, config.SECRET);
        req.loggedUserId = decoded.id; // save user ID and role into request object
        req.loggedUserRole = decoded.role;
        next();
    } catch (err) {
          return res.status(401).json({ success: false, msg: "Unauthorized!" });
    }
};
…
```

File *auth.controller.js* on *controllers* folder

You could also add functions to check for user role if they are not fully available on JWT payload, for example
(e.g.: from user ID, extract user info from DB and add it to request object)

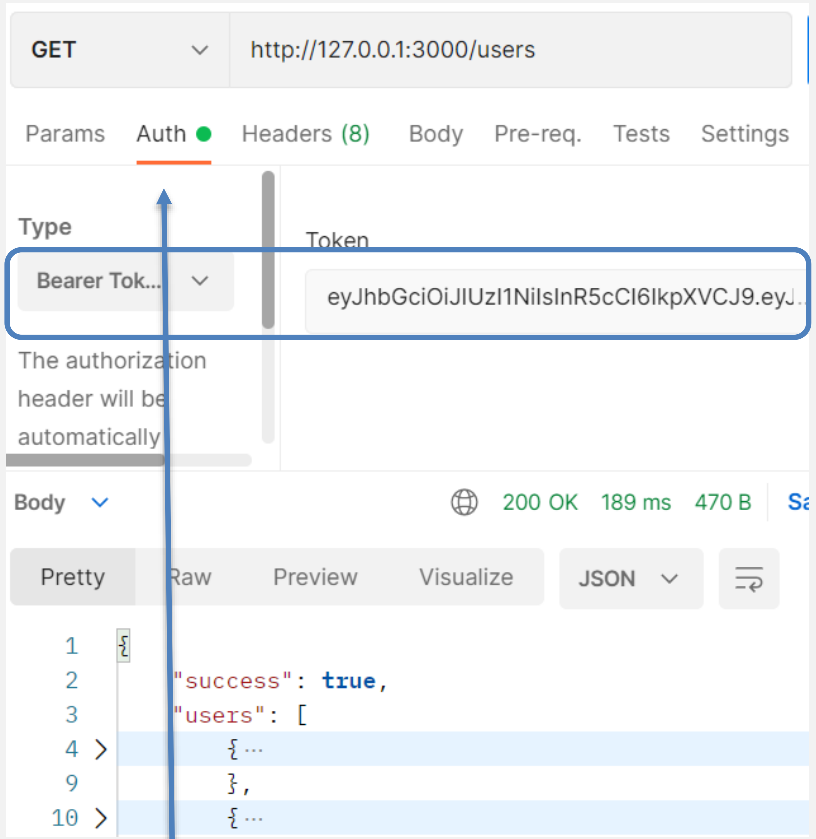
JWT Signup & Login example

- Add route handlers to list all users in `user.controller.js`: only available for administrators

File *user.controller.js* on *controllers* folder

```
exports.getAllUsers = async (req, res) => {
    try {
            if (req.loggedUserRole !== "admin")
            return res.status(403).json({
                success: false, msg: "This request requires ADMIN role!"
            });
        // do not expose users' sensitive data
        let users = await User.findAll({ attributes: ['id', 'username', 'email', 'role'] })
        res.status(200).json({ success: true, users: users });
    }
    catch (err) {
        res.status(500).json({
            success: false, msg: err.message || "Some error occurred while retrieving all users."
        });
    };
};
```

JWT Signup & Login example



GET http://127.0.0.1:3000/users

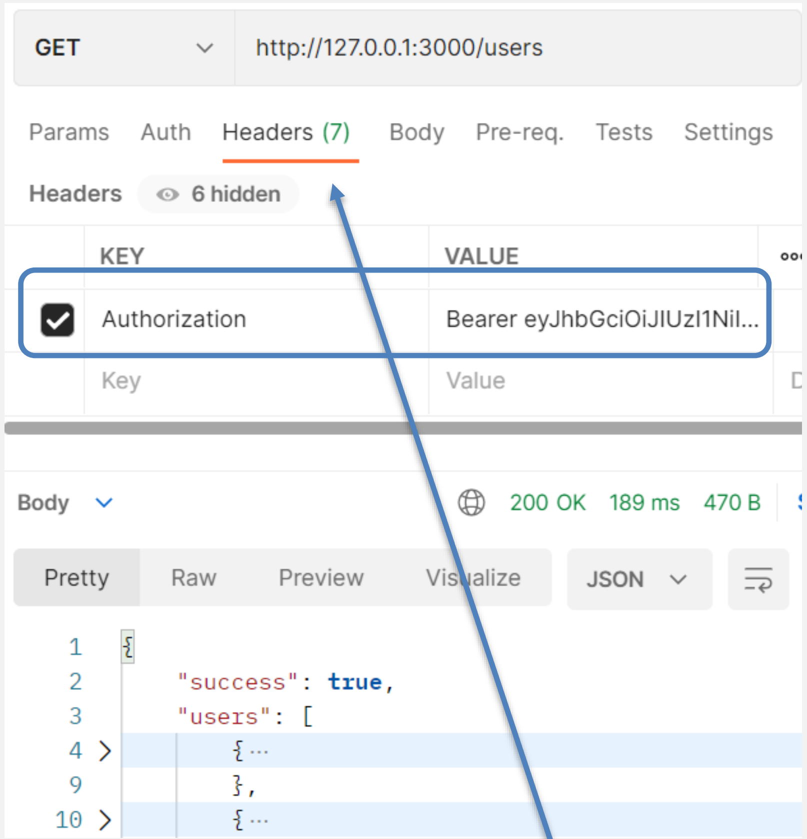
Type: Bearer Tok...

Token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ...

The authorization header will be automatically

200 OK 189 ms 470 B

```
{
    "success": true,
    "users": [
        {...
        },
        {...
```



GET http://127.0.0.1:3000/users

	KEY	VALUE
✓	Authorization	Bearer eyJhbGciOiJIUzI1NiI...
	Key	Value

200 OK 189 ms 470 B

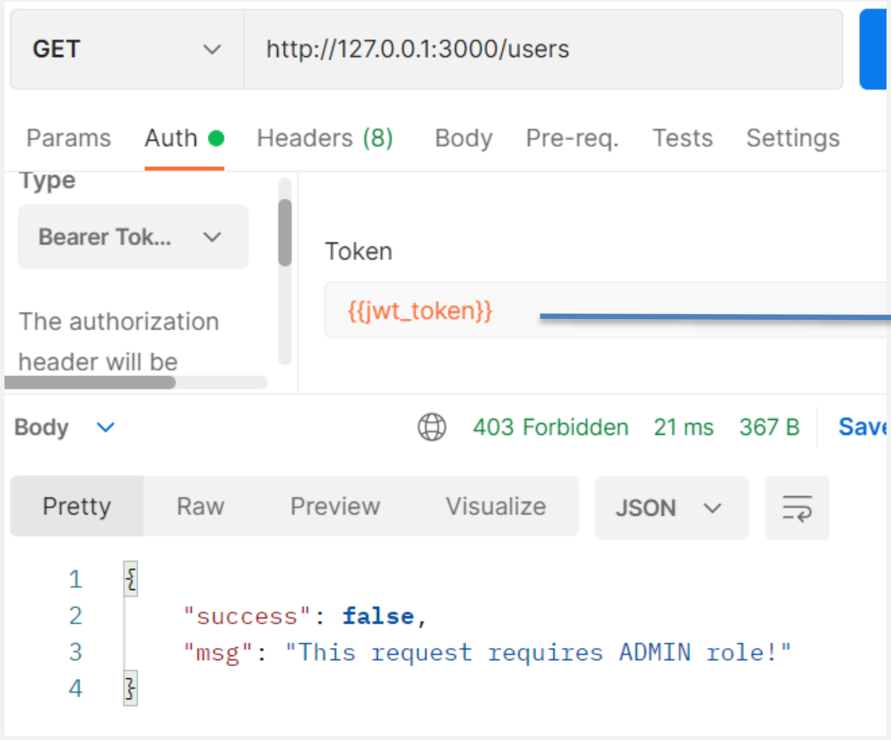
```
{
    "success": true,
    "users": [
        {...
        },
        {...
```

1) use Authorization helper

Two ways to access protected route with JWT token in authorization header using Postman
MUST READ

2) add Authorization header

JWT Signup & Login example



http://127.0.0.1:3000/users

The authorization header will be

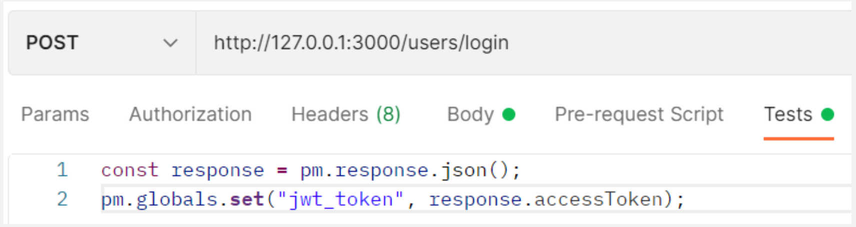
{{jwt_token}}

403 Forbidden 21 ms 367 B

```
{
    "success": false,
    "msg": "This request requires ADMIN role!"
}
```

Un-authorized access to ADMIN route

Use Postman environment or global **variables** to avoid copy-pasting of tokens between requests
MUST READ



http://127.0.0.1:3000/users/login

```
const response = pm.response.json();
pm.globals.set("jwt_token", response.accessToken);
```

JWT Signup & Login example

- Complete the API:
  - **Get one user** to show its full data: add route handler so that this route is only accessible for administrators or logged user
  - **Add/delete/update a comment**: routes only accessible for logged users; only the authors of a comment can delete or alter it