AdonisJS

What is AdonisJS?

Adonis JS is a Node framework to build a web app or api. The idea being that with just Adonis and it's included libraries you can build a website and not have to rely on a multitude of different frameworks. This guide is to walk you through creating an AdonisJS Application.

Setup

Step 1 Install NVM

If you haven't already installed nvm please do so. https://github.com/nvm-sh/nvm

Step 2 Install the NodeJS LTS that Adonis supports

Currently that's version 16. So once nvm is installed you can run
`nvm install 16`
Then
`nvm use 16`

Step 3 Create a project

Open your terminal and go to wherever you like to keep your projects. Then run
`npm init adonis-ts-app@latest forum_backend`

- select api
- enter forum_backend for the name
- y for eslint
- y for prettier

Now open up the project in your ide of choice(I prefer phpstorm or intellij)

Step 4 get docker

if you don’t already have docker desktop installed go to https://www.docker.com/products/docker-desktop then download an install it.

Docker is a way to “containerize” your applications so that all developers working on a project can get the same dependencies. It’s an interesting technology that is worth looking into.

Step 5 create Dockerfile

create a file called Dockerfile and put the following in it

```
FROM node:16
EXPOSE 3333
WORKDIR /app
VOLUME /app
CMD npm i && node ace migration:run && node ace serve --watch
```

Step 6 create docker-compose.yml file

With the Docker and Docker compose we can create containers for a database and a node backend

```
version: '3.8'
services:
    mysqldb:
        platform: linux/amd64
        image: mysql:5.7
        ports:
            - 3306:3306
        volumes:
            - .db:/var/lib/mysql
        environment:
            MYSQL_DATABASE: 'forum_backend'
            MYSQL_USER: 'forum_backend'
            MYSQL_PASSWORD: 'f9082m3e4f09'
            MYSQL_ROOT_PASSWORD: 'f9082m3e4f09'
            MYSQL_ROOT_HOST: '%'
            MYSQL_HOST: '%'
 backend:
        depends_on:
            - mysqldb
        ports:
            - 3333:3333
        build:
            dockerfile: ./Dockerfile
        stdin_open: true
        volumes:
            - .:/app
```

Step 7 Add Docker Ignore file

We need to ignore the node_modules otherwise native modules won't get built inside of the docker instance

create a file called .dockerignore

the only line inside of it should be

```
node_modules
```

Unfortunately, whenever you npm install a package you'll need to re-run docker compose to get it to load

Step 8 Install and configure lucid(Database library)

First install it
`npm i @adonisjs/lucid`

Now configure
`node ace configure @adonisjs/lucid`

Right now we're using mysql so select that option

Next open the instructions in the browser

Step 9 Add config information to env.ts file

If you opened those instructions then you'll notice it's asking you to add some stuff to the Env.rules object inside of env.ts so you should end up making something like this

```
export default Env.rules({
  HOST: Env.schema.string({ format: 'host' }),
  PORT: Env.schema.number(),
  APP_KEY: Env.schema.string(),
  APP_NAME: Env.schema.string(),
  DRIVE_DISK: Env.schema.enum(['local'] as const),
  NODE_ENV: Env.schema.enum(['development', 'production', 'testing'] as const),
})
```

turn into this

```
export default Env.rules({
 HOST: Env.schema.string({ format: 'host' }),
 PORT: Env.schema.number(),
 APP_KEY: Env.schema.string(),
 APP_NAME: Env.schema.string(),
 DRIVE_DISK: Env.schema.enum(['local'] as const),
 NODE_ENV: Env.schema.enum(['development', 'production', 'testing'] as const),

 DB_CONNECTION: Env.schema.string(),
 MYSQL_HOST: Env.schema.string({ format: 'host' }),
 MYSQL_PORT: Env.schema.number(),
 MYSQL_USER: Env.schema.string(),
 MYSQL_PASSWORD: Env.schema.string.optional(),
 MYSQL_DB_NAME: Env.schema.string(),
})
```

Step 10 add correct values to the .env file

Open up the .env file. Copy your APP_KEY value somewhere

Be Aware if you've changed any of the values in the docker files(for instance mysql_password) then some of these values might have changed. Otherwise your .env file should use these values

```
PORT=3333
HOST=0.0.0.0
NODE_ENV=development
APP_KEY=
DRIVE_DISK=local
DB_CONNECTION=mysql
MYSQL_HOST=mysqldb
MYSQL_PORT=3306
MYSQL_USER=forum_backend
MYSQL_PASSWORD=f9082m3e4f09
MYSQL_DB_NAME=forum_backend
```

Take your previous App key and paste it in there.

Were you silly and forgot to copy your original app key?
Then you can run
`node ace generate:key`
To generate a new one

Step 11 install and configure the auth module

install

```
npm i @adonisjs/auth
```

configure

```
node ace configure @adonisjs/auth
```

- Select Lucid
- Select API Tokens
- Enter User for the module name
- y to create the migration
- Database is the provider
- y to create the api_tokens migration

next we need to install the hash algorithm that adonis requires

```
npm install phc-argon2
```

Step 12 Now we need to add the auth middleware. Open up start/kernel.ts

Change

```
Server.middleware.registerNamed({});
```

To

```
Server.middleware.registerNamed({
    auth: () => import("App/Middleware/Auth")
});
```

Step 13 Install bouncer

Bouncer is a library to Authorize requests. It lets you write code to determine whether someone has “the right” to access something.

```
npm i @adonisjs/bouncer
```

Then run

```
node ace configure @adonisjs/bouncer
```

Step 14 update the migrations

Migrations are how we tell the database how it needs to update itself. When we setup the auth module we had it automatically make some migration files for us.

Since we’re using mysql 5.7 in our docker we have to update our migrations to have default values for our timestamps

inside of /database/migrations there should be 2 files. They start with a bunch of numbers then have _users.ts and _api_token.ts.

Open the _user.ts and change the created_at and updated_at columns by adding a default value

```
table.timestamp('created_at', { useTz: true }).notNullable().defaultTo(this.now())
table.timestamp('updated_at', { useTz: true }).notNullable().defaultTo(this.now())
```

In the _api_tokens.ts file change the created_at value to have a default value

```
table.timestamp('created_at', { useTz: true }).notNullable().defaultTo(this.now())
```

Any time you make a migration in the future you should just have to restart your docker compose to have it apply

Step 15 Add support for Decorators

Open up your tsconfig.json file

inside of “compilerOptions” add

```
"experimentalDecorators": true,
"emitDecoratorMetadata": true
```

Step 16 Change some styling rules

Obviously, these are my personal preferences so if you’re doing this for a personal project then you can set values you prefer or skip this step entirely. In case you were wanting to change something these are the prettier config values https://prettier.io/docs/en/options.html

Open up the .editorconfig file
change `indent_size = 2` to `indent_size = 4`

Open up the .prettierrc file and remove everything and make it look like this

```
{
    "trailingComma": "es5",
    "semi": true,
    "useTabs": false,
    "bracketSpacing": false,
    "arrowParens": "always",
    "printWidth": 120,
    "tabWidth": 4
}
```

Then run

`npm run format`

you can use that command any time you want to auto format your code

Step 17 Running Docker Compose

we need to delete our node_modules because phpc-argon2 is a native module and needs to be built inside the docker container

```
rm -rf ./node_modules
```

Now run Docker compose(this can take a few minutes to finish loading)

```
docker compose up
```

I had to run it twice to get it to work

Any time you make a new migration you'll have to rerun docker compose

Anytime you add a native node module you'll have to remove the node_modules folder and rerun docker compose(fortunately there are no more native module installs in this tutorial)

Step 18 Turn on CORS

inside of config/cors.ts set

```
origin: true,
```

to enable CORS(which lets other sites make AJAX requests to your site)

Actually doing something

Hopefully the setup wasn't too painful of a process for you. If you think it was hard imagine having to fumble your way through it like I did. Now we can actually get started building something with Adonis.

Registration

The first thing we can get started on is user registration. Since we installed the auth module we should already have a user modle. Now we just need a controller and a route

Create the UserController

```
node ace make:controller UserController -r
npm run format
```

Now open up ./app/Controllers/Http/UsersController.ts which was just created for us. Since we are only building an API we can delete the create and edit functions.

Now it should look something like

```
import { HttpContextContract } from '@ioc:Adonis/Core/HttpContext'

export default class UsersController {
   public async index({}: HttpContextContract) {}

   public async store({}: HttpContextContract) {}

   public async show({}: HttpContextContract) {}

   public async update({}: HttpContextContract) {}

   public async destroy({}: HttpContextContract) {}
}
```

Edit the store function

inside of the UsersController.ts file we need to edit the store function.

First we need to add the User model and the schema imports to the top of the file

```
import User from "App/Models/User";
import {schema} from "@ioc:Adonis/Core/Validator";
```

Then the store function should end up something like this

```
public async store({request, response}: HttpContextContract) {
    //create the schema
 const newUserSchema = schema.create({
  email: schema.string({trim: true}),
     password: schema.string(),
 });

 //validate the request with the schema
 //if it fails it will automatically send an error to the user
 const requestBody = await request.validate({schema:newUserSchema});

 //get the email and password
 const email = requestBody.email;
 const password = requestBody.password;

 //check to see if the user already exists
 //we don't want to try to create the same user twice
 const currentUser = await User.findBy("email", email);
 if (currentUser !== null) {
  response.badRequest("user already exists");
  return;
 }

    //populate a new user with data
    const newUser = new User();
    newUser.email = email;
    newUser.password = password;

    //save it to the database
    await newUser.save();

    //return the new user
    return newUser;
}
```

Add the routes

inside of start/routes.ts

add this route resource

```
Route.resource("users", "UsersController").apiOnly().except(["destroy", "update", "index"]);
```

Try it out

You should now be able to create a user by sending a post request with email and password to http://localhost:3333/users. You can use the Postman app to test your endpoints

Login

Next we need to be able to login. Lets go back to the UsersController.ts file we were working with before and create a login function

Add the login function to UsersController

```
public async login({request, auth}: HttpContextContract){
  //create the schema for this request
  const loginSchema = schema.create({
        email: schema.string({trim: true}),
        password: schema.string(),
  });

  //validate the request
  const userInfo = await request.validate({schema: loginSchema});

  //attempt to generate a token
  const token = await auth.use("api").attempt(userInfo.email, userInfo.password, {expiresIn: "
  const user = auth.user;
  return {token, user};
}
```

Add the route to the start/routes.ts file

We also need to add a login route. Go to your start/routes.ts file and add this at the top.

```
import UsersController from "App/Controllers/Http/UsersController";
```

Then add

```
Route.post("/users/login", (ctx) => {
        return new UsersController().login(ctx);
});
```

Try it

you can make a post request to http://localhost:3333/users/login with an email and password to generate a token

Get a user by ID

We need to return a user based on their ID, but we don't necessarily want other users to view someone else's information. This time we'll need to use Bouncer to stop unauthorized access.

Bouncer

open up ./start/bouncer.ts

At the top of the file we need to import User

```
import User from "App/Models/User";
```

Then we need to add a new action called viewUser.

Where it says

```
export const { actions } = Bouncer
```

Change it to

```
export const {actions} = Bouncer.define("viewUser", (user: User, userToView: User) => {
    return user.id === userToView.id;
});
```

This is saying that the logged in user and only access user info if they happen to be that user themselves.

UsersController

Open the Users controller back up. We need to change the code for the show method

```
public async show({params, bouncer, response, auth}: HttpContextContract) {
  //make sure id was passed into the params
  if (params.id === null) {
        return response.badRequest("id expected in params");
  }
  //this will check the Authorization header to see if the user is authenticated or not
  await auth.use("api").authenticate();
  //this will get the user or fail if the id is invalid
  const userToView = await User.findOrFail(params.id);
  //we use bouncer to determine that the logged in user can view this information
  await bouncer.authorize("viewUser", userToView);
  //lastly we can return the user
  return userToView;
}
```

Try it

You should now be able to try http://localhost:3333/users/1 or whatever id you want. Remember you need to set the Authorization header token with the token generated from login. If you're using Postman to test your endpoints you go to the Authorization tab, set type to Bearer token, then put in the actual token

Create a Topic

With the user stuff out of the way lets move on to Topics.

Make the model, migration, and controller

Thankfully, Adonis comes with a way to auto generate some of this for us

```
node ace make:model Topic -m -c
```

Update the migration

inside of ./database/migrations there should be a file that have a bunch of numbers follow by _topics.ts. Open up that file

```
 public async up() {
  this.schema.createTable(this.tableName, (table) => {
  //create an auto incremented id
  table.increments("id");
  //create a name column
  table.string("name").unique().notNullable();
  //create a column for description
  table.text("description").notNullable();
  //create a reference to the user table
  table.integer("user_id").unsigned().references("users.id").onDelete("RESTRICT").notNullable(
  table.timestamp("created_at", {useTz: true}).defaultTo(this.now());
  table.timestamp("updated_at", {useTz: true}).defaultTo(this.now());
  });
}
```

Since we made a migration you'll have to restart docker compose in order for that to be made for you

```
docker compose up
```

Update the Model

Open up the ./app/Models/Topic.ts file

```
import {DateTime} from "luxon";
import {BaseModel, BelongsTo, belongsTo, column} from "@ioc:Adonis/Lucid/Orm";
import User from "App/Models/User";

export default class Topic extends BaseModel {
    @column({isPrimary: true})
    public id: number;

  @column()
    public name: string;

  @column()
    public description: string;

  @column()
    public userId: number;

  @column.dateTime({autoCreate: true})
    public createdAt: DateTime;

  @column.dateTime({autoCreate: true, autoUpdate: true})
    public updatedAt: DateTime;

  @belongsTo(() => User, {serializeAs: "user"})
    public user: BelongsTo<typeof User>;
}
```

Update the Controller

Open up the ./app/Controllers/TopicsController.ts file. We need to update the store function.

```
import {HttpContextContract} from "@ioc:Adonis/Core/HttpContext";
import {schema, rules} from "@ioc:Adonis/Core/Validator";
import Topic from "App/Models/Topic";

export default class TopicsController {
    public async index({}: HttpContextContract) {}

    public async store({request, auth}: HttpContextContract) {
        //create our schema we want a name and a description
        const topicsSchema = schema.create({
         name: schema.string({}, [rules.maxLength(50),rules.minLength(3)]),
         description: schema.string({}, [rules.maxLength(500), rules.minLength(3)]),
        });
        //validate the schema
        const payload = await request.validate({schema: topicsSchema});
        const newTopic = new Topic();
        newTopic.name = payload.name;
        newTopic.description = payload.description;
        //associate the topic with the user that is creating it
        await newTopic.related("user").associate(auth.user!);
        //save the new topic to the database
        await newTopic.save();
        //load in the user information
        await newTopic.load("user");
        return newTopic;
    }

    public async show({}: HttpContextContract) {}

    public async update({}: HttpContextContract) {}

    public async destroy({}: HttpContextContract) {}
}
```

Update the routes

Open up the /start/routes.ts file. We need to add

```
Route.resource("topics", "TopicsController").apiOnly().middleware({"*": "auth"});
```

This creates the api routes and requires authentication for all routes

Try it

if you make a post request to http://localhost:3333/topics with a name and a description you should be able to create topics.

don’t forget to add your auth token in the Authorization section

Get all Topics

Now lets work on our endpoint to get a list of all the topics

Update the TopicsController

Open up the /app/Controllers/Http/TopicsController.ts file. Fortunately, now that we’ve already done most of the work with setting up the model, migration, and routes it will be easier.

We need to update the index function

```
public async index({}: HttpContextContract) {
    const topics = await Topic.query().preload("user").orderBy("created_at");
    return topics;
}
```

Try it

make a get request to http://localhost:3333/topics. Don’t forget your auth token.

Get A Specific Topic

Eventually we’ll have to update this as we add the threads, but for now let’s open up the TopicsController and look at the show function.

Update the show function

```
public async show({params, response}: HttpContextContract) {
    const topic = await Topic.query()
        .preload("user")
        .preload("threads", (query) => {
            //load in users with our threads
         query.preload("user");
     })
        .where("id", params.id);
     if (topic.length <= 0) {
        response.notFound("No Topic by that id exists");
       return;
     }
     return topic[0];
}
```

Update a Topic

Now lets take a look at the update function

Create new bouncer action

In the /start/bouncer.ts file we need to add the topic model as an import

```
import Topic from "App/Models/Topic";
```

Next we need to add the updateTopic action to the bouncer actions. So this

```
export const {actions} = Bouncer.define("viewUser", (user: User, userToView: User) => {
    return user.id === userToView.id;
});
```

needs to turn into this

```
export const {actions} = Bouncer.define("viewUser", (user: User, userToView: User) => {
    return user.id === userToView.id;
}).define("updateTopic", (user: User, topic: Topic) => {
    return user.id === topic.userId;
});
```

Basically saying that you can only update the topic if you are the user that made the topic

Update the update function

The TopicsController update function should turn into this

```
public async update({bouncer, request, params}: HttpContextContract) {
  //create schema because we are updating both name and description are optional
  const updateSchema = schema.create({
        name: schema.string.optional({trim: true}, [rules.maxLength(50), rules.minLength(3)]),
  description: schema.string.optional({trim: true}, [rules.maxLength(500), rules.minLength(3)]
  });
  const updatePayload = await request.validate({schema: updateSchema});
  //get the topic
  const topic = await Topic.findOrFail(params.id);
  //verify the user made the topic
  await bouncer.authorize("updateTopic", topic);
  //update the values
  topic.name = updatePayload.name ?? topic.name;
  topic.description = updatePayload.description ?? topic.description;
  //save the topic
  await topic.save();
  //load in the user
  await topic.load("user");
  return topic;
}
```

Try it

make a put request to http://localhost:3333/topics/1 (or whatever ID you have) and pass in name and description. Don't forget your auth token!

Delete a Topic

To delete a topic we'll have to update the controller and bouncer actions similar to the update function.

Create new bouncer action

in /start/bouncer.ts define a new action deleteTopic

```
export const {actions} = Bouncer.define("viewUser", (user: User, userToView: User) => {
    return user.id === userToView.id;
})
    .define("updateTopic", (user: User, topic: Topic) => {
        return user.id === topic.userId;
  })
    .define("deleteTopic", (user: User, topic: Topic) => {
        return user.id === topic.userId;
  });
```

Update the destroy function

Open up the TopicsController and update the destroy function

```
public async destroy({params, bouncer}: HttpContextContract) {
 let topic = await Topic.findOrFail(params.id);
 await bouncer.authorize("deleteTopic", topic);
 await topic.delete();
 return topic;
}
```

Try it

make a delete request to http://localhost:3333/topics/1 (or whatever ID you have). Don't forget your auth token!

Create a Thread

Now we move on to Threads. We'll be doing a lot of similar things to Topics so it shouldn't be too confusing.

Create Model, Migration, and Controller

Lets have adonis generate these files for us

```
node ace make:model Thread -m -c
```

Update the Migration

open up /database/migrations/SOME_NUMBERS_thread.ts and update the up function

```
public async up() {
  this.schema.createTable(this.tableName, (table) => {
    table.increments("id");
    table.string("name").notNullable();
    //these add foreign keys for users and topics
    table.integer("user_id").unsigned().notNullable().references("users.id");
    table.integer("topic_id").unsigned().notNullable().references("topics.id");
    table.timestamp("created_at", {useTz: true}).defaultTo(this.now());
    table.timestamp("updated_at", {useTz: true}).defaultTo(this.now());
  });
}
```

Remember now that you've made a migration you'll need to restart docker compose to get it to update then run

```
docker compose up
```

Update the Thread model

Open up the Thread model

```
import {DateTime} from "luxon";
import {BaseModel, BelongsTo, belongsTo, column} from "@ioc:Adonis/Lucid/Orm";
import User from "App/Models/User";
import Topic from "App/Models/Topic";

export default class Thread extends BaseModel {
  @column({isPrimary: true})
  public id: number;

  @column()
  public name: string;

  @column()
  public userId: number;

  @column()
  public topicId: number;

  @column.dateTime({autoCreate: true})
  public createdAt: DateTime;

  @column.dateTime({autoCreate: true, autoUpdate: true})
  public updatedAt: DateTime;

  @belongsTo(() => User)
  public user: BelongsTo<typeof User>;

  @belongsTo(() => Topic)
  public topic: BelongsTo<typeof Topic>;
}
```

Update the Topic Model

Open up the Topic model. Because a Topic can have many threads, we need to update the model to reflect this

Add

```
@hasMany(() => Thread)
public threads: HasMany<typeof Thread>;
```

To the Topic model class and be sure to update your imports

Update the ThreadsController

Open up the ThreadsController

```
import {HttpContextContract} from "@ioc:Adonis/Core/HttpContext";
import {rules, schema} from "@ioc:Adonis/Core/Validator";
import Topic from "App/Models/Topic";
import Thread from "App/Models/Thread";

export default class ThreadsController {
    public async index({}: HttpContextContract) {}

   public async store({request, auth}: HttpContextContract) {
     //threads require a name a topicId
     const threadSchema = schema.create({
       name: schema.string({trim: true}, [rules.minLength(5), rules.maxLength(70)]),
      topicId: schema.number(),
    });
     const threadPayload = await request.validate({schema: threadSchema});
    //make sure the topicId belongs to an actual topic
    const topic = await Topic.findOrFail(threadPayload.topicId);
    const thread = new Thread();
    thread.name = threadPayload.name;
    //this is required because the associate functions below
    //try to insert to the database
    //even though it shouldn't
    thread.userId = auth.user!.id;
    //associate the thread with the topic
    await thread.related("topic").associate(topic);
    //associate the thread with the user that is creating it
    await thread.related("user").associate(auth.user!);
    //save the thread
    await thread.save();
    //load in relations
    await thread.load("user");
    await thread.load("topic");
    return thread;
}

   public async show({}: HttpContextContract) {}

   public async update({}: HttpContextContract) {}

   public async destroy({}: HttpContextContract) {}
}
```

Update the Routes

in the routes.ts file we need to add the thread routes

```
Route.resource("threads", "ThreadsController").apiOnly().except(["index"]).middleware({"*": "a
```

Try it

make a post request to http://localhost:3333/threads with name and a valid topicId. Don’t forget your auth token!

Update Your Topic Model and Controller

We need to give the Topic model a “has many” relationship to Threads.

Topic Model

Open up the Topic Model and add

```
@hasMany(() => Thread)
public threads: HasMany<typeof Thread>;
```

Update TopicsController

We need to update the store function on the TopicsController to load in the threads

```
public async show({params}: HttpContextContract) {
    const topic = await Topic.findOrFail(params.id);
    await topic.load("user");
    await topic.load("threads");
    return topic;
}
```

Now when you make a GET request to an individual topic it will include the threads with it

Get a Thread

Lets move on to getting a thread

Update the ThreadsController

We need to update the show function inside of the ThreadsController

```
public async show({params}: HttpContextContract) {
  const thread = await Thread.findOrFail(params.id);
  await thread.load("user");
  await thread.load("topic");
  return thread;
}
```

Try it

make a get request to http://localhost:3333/threads/1 (or whatever id). Don't forget your auth token.

Update a Thread

Now lets move on to updating a thread

Update Bouncer

open /start/bouncer.ts

add the thread model to the top

```
import Thread from "App/Models/Thread";
```

Next add the "updateThread" action to the actions

```
export const {actions} = Bouncer.define("viewUser", (user: User, userToView: User) => {
    return user.id === userToView.id;
})
    .define("updateTopic", (user: User, topic: Topic) => {
        return user.id === topic.userId;
  })
    .define("deleteTopic", (user: User, topic: Topic) => {
        return user.id === topic.userId;
  })
    .define("updateThread", (user: User, thread: Thread) => {
        return user.id === thread.userId;
  });
```

Update the Threads Controller

Lets update the "update" function inside the ThreadsController

```
public async update({request, params, bouncer}: HttpContextContract) {
  //get the thread
  const thread = await Thread.findOrFail(params.id);
  const threadSchema = schema.create({
        name: schema.string({trim: true}, [rules.minLength(5), rules.maxLength(70)]),
  });
  const threadPayload = await request.validate({schema: threadSchema});
  await bouncer.authorize("updateThread", thread);
  thread.name = threadPayload.name;
  await thread.save();
  return thread;
}
```

Try it

make a put request to http://localhost:3333/threads/1 (or whatever ID you have) and pass in name. Don't forget your auth token!

Deleting a Thread

Let's move on to deleting a thread

Update Bouncer

We need to add a new bouncer action "deleteThread."

```
export const {actions} = Bouncer.define("viewUser", (user: User, userToView: User) => {
    return user.id === userToView.id;
})
.define("updateTopic", (user: User, topic: Topic) => {
        return user.id === topic.userId;
})
.define("deleteTopic", (user: User, topic: Topic) => {
    return user.id === topic.userId;
})
.define("updateThread", (user: User, thread: Thread) => {
     return user.id === thread.userId;
})
.define("deleteThread", (user: User, thread: Thread) => {
    return user.id === thread.userId;
});
```

Update ThreadsController

We need to update the destroy function inside of ThreadsController

```
public async destroy({params, bouncer}: HttpContextContract) {
  const thread = await Thread.findOrFail(params.id);
  await bouncer.authorize("deleteThread", thread);
  await thread.delete();
  return thread;
}
```

Try it

make a delete request to http://localhost:3333/threads/1 (or whatever ID you have). Don't forget your auth token!

Creating a Post

We're almost done, just need to Implement all the CRUD for Posts!

Generate Model, Migration, and Controller for Posts

```
node ace make:model Post -c -m
```

Update the Migration

Open up the /database/migrations/SOME_NUMBERS.posts.ts file and update the up function

```
public async up() {
    this.schema.createTable(this.tableName, (table) => {
        table.increments("id");
        table.text("text").notNullable();
        table.integer("thread_id").unsigned().notNullable().references("threads.id").onDelete(
        table.integer("user_id").unsigned().notNullable().references("users.id").onDelete("CAS
        table.timestamp("created_at", {useTz: true}).defaultTo(this.now());
        table.timestamp("updated_at", {useTz: true}).defaultTo(this.now());
  });
}
```

Remember for this migration to take effect you must restart docker compose

```
docker compose up
```

Update the Model

Open up the Post model

```
import {DateTime} from "luxon";
import {BaseModel, BelongsTo, belongsTo, column} from "@ioc:Adonis/Lucid/Orm";
import User from "App/Models/User";
import Thread from "App/Models/Thread";

export default class Post extends BaseModel {
  @column({isPrimary: true})
  public id: number;

  @column()
  public text: string;

  @column()
  public userId: number;

  @column()
  public threadId: number;

  @column.dateTime({autoCreate: true})
  public createdAt: DateTime;

  @column.dateTime({autoCreate: true, autoUpdate: true})
  public updatedAt: DateTime;

  @belongsTo(() => User)
  public user: BelongsTo<typeof User>;

  @belongsTo(() => Thread)
  public thread: BelongsTo<typeof Thread>;
}
```

Update the Controller

Open up the Post controller

```
import {HttpContextContract} from "@ioc:Adonis/Core/HttpContext";
import {rules, schema} from "@ioc:Adonis/Core/Validator";
import Thread from "App/Models/Thread";
import Post from "App/Models/Post";

export default class PostsController {

    public async store({request, auth}: HttpContextContract) {
        const postSchema = schema.create({
            text: schema.string({trim: true}, [rules.minLength(5), rules.maxLength(1000)]),
            threadId: schema.number(),
        });
        const postPayload = await request.validate({schema: postSchema});
        const thread = await Thread.findOrFail(postPayload.threadId);
        const post = new Post();
        post.text = postPayload.text;
        post.threadId = thread.id;
        await post.related("user").associate(auth.user!);
        await post.save();
        await post.load("user");
        return post;
  }

  public async show({}: HttpContextContract) {}

  public async update({}: HttpContextContract) {}

  public async destroy({}: HttpContextContract) {}
}
```

Add the Routes

Open up the /start/routes.ts file and add

```
Route.resource("posts", "PostsController").apiOnly().except(["index"]).middleware({"*": "auth"
```

Try it

make a post request to http://localhost:3333/posts with text and a valid threadId. Don't forget your auth token!

Update the Thread Model and Controller

Now that we have posts we need to update the Thread model and Controller with a "has many" relationship

Update Thread model

Open the Thread model

at the top add the import

```
import Post from "App/Models/Post";
```

Then inside of the class add

```
@hasMany(() => Post)
public posts: HasMany<typeof Post>;
```

You'll have to update your imports to include hasMany and HasMany

Update the ThreadsController

update the show function to load in the Posts

```
public async show({params, response}: HttpContextContract) {
  //we use the preload functions to load in our relationships
  //since we want the posts to also include our users
  //the post preload shows how to do that
  const thread =   await Thread.query()
        .preload("user")
        .preload("topic")
        .preload("posts", (builder) => {
            builder.preload("user");
        })
        .where("threads.id", params.id)
        .first();
  //if there is no thread then throw a 404 error
  if (thread === null) {
      response.notFound(`no thread with id ${params.id} exists`);
      return;
  }

  return thread;
}
```

Get a Post

Let's work on getting an individual post now

Update the PostsController

update the show function of the PostsController

```
public async show({params}: HttpContextContract) {
  const post = await Post.findOrFail(params.id);
  post.load("user");
  post.load("thread");
  return post;
}
```

Try it

make a get request to http://localhost:3333/posts/1 (or whatever id). Don't forget your auth token!

Update a Post

Update Bouncer

in the /start/bouncer.ts file import the Post model

```
import Post from "App/Models/Post";
```

now add an action "updatePost"

```
export const {actions} = Bouncer.define("viewUser", (user: User, userToView: User) => {
    return user.id === userToView.id;
})
  .define("updateTopic", (user: User, topic: Topic) => {
        return user.id === topic.userId;
  })
  .define("deleteTopic", (user: User, topic: Topic) => {
        return user.id === topic.userId;
  })
  .define("updateThread", (user: User, thread: Thread) => {
        return user.id === thread.userId;
  })
  .define("deleteThread", (user: User, thread: Thread) => {
        return user.id === thread.userId;
  })
  .define("updatePost", (user: User, post: Post) => {
        return user.id === post.userId;
  });
```

Update PostsController

Open the PostsController and update the update function

```
public async update({bouncer, params, request}: HttpContextContract) {
  const postSchema = schema.create({
        text: schema.string({trim: true}, [rules.minLength(5), rules.maxLength(500)]),
  });
  const postPayload = await request.validate({schema: postSchema});
  const post = await Post.findOrFail(params.id);
  await bouncer.authorize("updatePost", post);
  post.text = postPayload.text;
  await post.save();
  return post;
}
```

Try it

make a put request to http://localhost:3333/posts/1 (or whatever ID you have) and pass in text. Don't forget your auth token!

Delete a Post

last but not least it's time to delete a post

Update Bouncer

Add an action for "deletePost"

```
export const {actions} = Bouncer.define("viewUser", (user: User, userToView: User) => {
    return user.id === userToView.id;
})
  .define("updateTopic", (user: User, topic: Topic) => {
        return user.id === topic.userId;
  })
  .define("deleteTopic", (user: User, topic: Topic) => {
        return user.id === topic.userId;
  })
  .define("updateThread", (user: User, thread: Thread) => {
        return user.id === thread.userId;
  })
  .define("deleteThread", (user: User, thread: Thread) => {
        return user.id === thread.userId;
  })
  .define("updatePost", (user: User, post: Post) => {
        return user.id === post.userId;
  })
  .define("deletePost", (user: User, post: Post) => {
        return user.id === post.userId;
  });
```

Update PostsController

update the destroy function of PostsController

```
public async destroy({params, bouncer}: HttpContextContract) {
    const post = await Post.findOrFail(params.id);
    await bouncer.authorize("deletePost", post);
    await post.delete();
    return post;
}
```

Try it

make a delete request to http://localhost:3333/posts/1 (or whatever ID you have). Don’t forget your auth token!