**Basic Authentication with a .NET Core Web API**

# Authentication

## **Introduction**

Welcome to the authentication section.

In this section, we will add users to our role-playing game. Users will be able to create an account and add RPG characters to that account.

So, with that, we will also introduce JSON Web Tokens. These tokens will be used to authenticate the user, meaning we’re making sure that this particular user is allowed to select certain characters, update them, and so on.

Additionally, with users and their characters, we already introduce the first relation in our database. One user can have several RPG characters.

But before we get to the details, let’s start with the new user model

## **The User Model**

To add the user model, we create a new C# class in the Models folder and call this class User.

Let’s keep this User model simple. We give the user an Id of type int, a Username of type string, and we need a password, of course. But this won’t just be a password string. We actually want to save a hash value of the password and we also need a salt to create a unique password hash. And these two properties will be byte arrays.

So, let’s add two byte arrays, the PasswordHash and the PasswordSalt.

public class User

{

public int Id { get; set; }

public string Username { get; set; }

public byte[] PasswordHash { get; set; }

public byte[] PasswordSalt { get; set; }

}

That’s it for now. Let’s run another migration to see the new User model in the database.

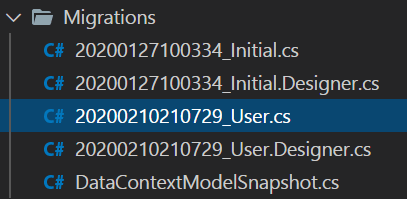
Remember, we have to add another DbSet<> so that Entity Framework knows what tables to create.

We add the new DbSet<User> in the DataContext class.

public DbSet<User> Users { get; set; }

Again, we simply pluralize the type User to get a suitable table name.

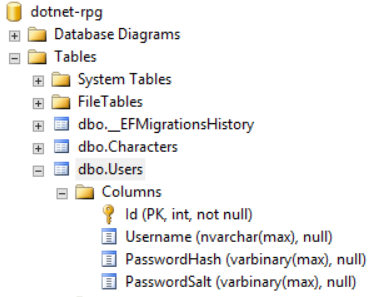
Now we can run the migration. First, in the terminal, we enter dotnet ef migrations add User to add the User to the migration.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--ZwgPT0s0--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/8esgm7urs7vop31ktxtj.png)

In the file explorer, we find our new migration. When we have a look at the migration file, we see the Up() and Down() methods again. Similar to the migration of the RPG character, a corresponding database table will be created with all its fields and the Id as the primary key.

Let’s add this table with dotnet ef database update.

After the successful update, we can see the Users table in our database

[](https://res.cloudinary.com/practicaldev/image/fetch/s--Dz_ynItu--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/9l5mfvvczkj6ho53656o.png)

But what about a relation? Users should be able to have several RPG characters.

Let’s add that relation now.

## **First Relation**

There are three types of relations in a relational database, one-to-one, one-to-many and many-to-many.

What does that mean?

A one-to-one relation is pretty simple. In our example, this would mean that a user would only have one RPG character. And this RPG character only has this one particular user. These two entities are linked and when you know the user, you can also identify the character and vice versa. The user cannot have another character. The same states for the character.

A one-to-many relationship is what we want to implement. A user can have several RPG characters. But the characters are only linked to that single user. One user, many characters.

If the same characters would be linked to different users, we would have a many-to-many relationship. In essence, this would mean that RPG characters are shared amongst several users. This could be an interesting concept, but we’ll stick to the one-to-many relationship.

More suitable for a many-to-many relationship would be a relation between characters and skills. For instance, several magicians could have several skills like throwing fireballs or summoning creatures. We’ll do something like that later.

To realize the one-to-many relationship between users and characters, we add properties to our models.

First, the User gets a List<> of RPG characters.

public List<Character> Characters { get; set; }

Actually, that’s already sufficient. That way we’re able to select all characters of a particular user.

But what about the other way around? We have an RPG character and want to select the corresponding user? To be able to do that, we add a User property to the Character model, as well.

public User User { get; set; }

Now we’ve got our relationship. Let’s add a migration again with dotnet ef migrations add UserCharacterRelation.

The Up() method is different now. It adds a new column UserId to the Characters table and also defines an index and a foreign key to the column Id of the Users table. That’s how the relationship is defined in the database.

protected override void Up(MigrationBuilder migrationBuilder)

{

migrationBuilder.AddColumn<int>(

name: "UserId",

table: "Characters",

nullable: true);

migrationBuilder.CreateIndex(

name: "IX\_Characters\_UserId",

table: "Characters",

column: "UserId");

migrationBuilder.AddForeignKey(

name: "FK\_Characters\_Users\_UserId",

table: "Characters",

column: "UserId",

principalTable: "Users",

principalColumn: "Id",

onDelete: ReferentialAction.Restrict);

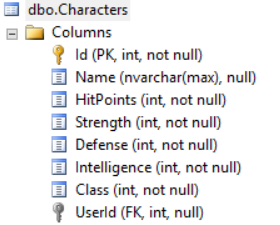
}

It’s interesting to mention, that onDelete is set to ReferentialAction.Restrict. This means that you can’t delete a user if he or she still has a character. Another option would be Cascade. In that case, all characters of the user will be deleted, if the user would be deleted.

The Down() method looks pretty straight forward. It would just revert the actions of the Up() method, meaning removing the foreign key, the index, and the new column.

Let’s update the database now with dotnet ef database update.

You see, our Characters now have a UserId and we’re able to create some relations.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--HXBLrVs5--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/7yy9dmxliotbzwywppbx.png)

But before we let users create any RPG characters, let’s implement the authentication first.

## **Authentication Theory**

So, how would the authentication work in general?

You have already seen that we want to store the password of the user as a hashed value. The other option would be to store the password in plain text and when the user is trying to log in, we just compare the entered value with the stored value. But that's absolutely not recommended. Even big websites get hacked, database leaks happen, it's just not secure at all. And you would see all the passwords of your users. I guess they wouldn't like that at all.

That's why we're using a hash value. But even hashing a password is not enough.

To hash a password we're using a particular cryptography algorithm like HMACSHA512, for instance. That algorithm is always the same, hence the hash of a certain password is also always the same.

So, you would see if users use the same passwords and, much worse, there are tools that can revert these algorithms and so receive the plain password in seconds via brute force.

And that's why we're also using a password salt. A salt can be a random sequence of letters and numbers that is used together with the actual password.

The algorithm gets the password and a salt as input so that it produces different hash values even with the same password.

To be able to get the correct result, we store the password salt in the database.

How does this look in the end?

When the user registers with a password, we grab this password, generate a random salt, throw both into the algorithm and get a beautiful final hash value.

When the user wants to log in with that password, we actually do the same, except grabbing the stored salt from the database. The resulting hash value will be compared to the stored hash value and if both values are the same, the user is successfully authenticated.

So far the theory. Let's build that now.

## **Authentication Repository**

For the implementation of the authentication, we're using the repository pattern. You already know it a bit from our DataContext and we’re actually using this pattern already with our CharacterService. Since we’re using this again now, I think it’s time to explain that a bit.

The documentation from Microsoft describes repositories pretty well.

Repositories are classes or components that encapsulate the logic required to access data sources. They centralize common data access functionality, providing better maintainability and decoupling the infrastructure or technology used to access databases from the domain model layer. If you use an Object-Relational Mapper (ORM) like Entity Framework, the code that must be implemented is simplified, thanks to LINQ and strong typing. This lets you focus on the data persistence logic rather than on data access plumbing.

Well, this is very technical, so let’s break this down a little.

Thanks to Entity Framework, we don’t have to write the SQL statements ourselves and we don’t have to worry about setting up the database connection every time we want to access the entities of the database.

We just use Entity Framework to do all this with the help of the DbContext. So, we inject the DataContext into a C# class and we’re already able to access the database. If anything changes in the database or we want to use another database, we still use the DataContext to run all operations. We don’t have to change our code!

And that’s already the repository pattern in essence. We already do the same with the CharacterService. This service or repository enables our CharacterController to access all RPG character operations by injecting the ICharacterService interface. Any other controller could do the same. And if the code in the CharacterService methods changes, we don’t have to touch the controllers, because they just access the methods via the injected class.

We register the repository as a service in the Startup class in the ConfigureServices() method. Here we’re telling our application which implementation should be used if a service wants to inject a certain interface.

This means, we could create a CharacterService2, change the single line in the Startup class and everything works just fine with the new implementation.

That’s the magic of dependency injection and the repository pattern.

Enough with the theory.

Let’s create an interface and a C# class for authentication.

In the Data folder we create the interface IAuthRepository and also the class AuthRepository which implements the interface, of course.

public class AuthRepository : IAuthRepository

{

}

The interface gets three methods. The first is Register() with a User and a string as parameter and returning an int - the Id of the user, second method is Login() with two string parameters and returning a token as string, and the last method is UserExists() with the username as string to check if the user already exists. Notice that the last method does not return a ServiceResponse. This would be a bit over the top.

public interface IAuthRepository

{

Task<ServiceResponse<int>> Register(User user, string password);

Task<ServiceResponse<string>> Login(string username, string password);

Task<bool> UserExists(string username);

}

Alright, let’s implement the methods by using the quick fix menu and then let’s start with the user registration.

## **User Registration**

Registering a new user is nothing else than creating a new user in the database.

That’s not the whole magic, of course, but let’s start with this idea. To access the database, we need the DataContext again. So we add the constructor for the AuthRepository and inject the DataContext.

private readonly DataContext \_context;

public AuthRepository(DataContext context)

{

\_context = context;

}

Next, in the Register() method, we could simply access the \_context, add this new user, save all changes, create the ServiceResponse and send the new Id of the user back, because adding the new user generates a new id.

public async Task<ServiceResponse<int>> Register(User user, string password)

{

await \_context.Users.AddAsync(user);

await \_context.SaveChangesAsync();

ServiceResponse<int> response = new ServiceResponse<int>();

response.Data = user.Id;

return response;

}

Do you see the problem with this? What's up with the password?

If we would just use the given password string like that, it would be stored in plain text in the database. Not good.

So, let’s write a method to hash the password.

At the bottom, we create a private method CreatePasswordHash with the parameter password and also two out parameters passwordHash and passwordSalt. These last two will be stored in the database.

Now it’s getting interesting. We’re using the HMACSHA512 cryptography algorithm to generate a key and create the password hash. It can be found in the System.Security.Cryptography namespace.

Creating an instance of this class already generates a key that can be used as password salt. So, our passwordSalt is the hmac.Key.

After that, we use the class to generate a hash with the method ComputeHash() which takes the given password as bytes.

private void CreatePasswordHash(string password, out byte[] passwordHash, out byte[] passwordSalt)

{

using (var hmac = new System.Security.Cryptography.HMACSHA512())

{

passwordSalt = hmac.Key;

passwordHash = hmac.ComputeHash(System.Text.Encoding.UTF8.GetBytes(password));

}

}

That’s it. With the help of the out keywords we don’t need to return anything, the hash and salt are already available in the Register() method. So, let’s use this new method now.

At the top of the Register() method, we call CreatePasswordHash() with the given password and the new byte[] variables passwordHash and passwordSalt.

Then we set the hash and salt of the user object.

public async Task<ServiceResponse<int>> Register(User user, string password)

{

CreatePasswordHash(password, out byte[] passwordHash, out byte[] passwordSalt);

user.PasswordHash = passwordHash;

user.PasswordSalt = passwordSalt;

await \_context.Users.AddAsync(user);

await \_context.SaveChangesAsync();

ServiceResponse<int> response = new ServiceResponse<int>();

response.Data = user.Id;

return response;

}

We’re almost done. Remember, we have to check if the user already exists. That’s what we do next.

## **“User already exists.”**

The implementation is pretty straight forward.

We add the async keyword to the method and then start with an if statement where we access the \_context and check if a user with the same username already exists by calling the method AnyAsync() followed by a lambda expression. We need the Microsoft.EntityFrameworkCore using directive for that. We also turn the usernames to lowercase.

If the user already exists, we return true, in the other case, we return false. Simple as that.

public async Task<bool> UserExists(string username)

{

if (await \_context.Users.AnyAsync(x => x.Username.ToLower() == username.ToLower()))

{

return true;

}

return false;

}

You see, using a ServiceResponse in this method would be a bit over the top. We could even use a private method in this case. But maybe we need the method in a controller someday, so it doesn’t hurt to make this method publicly available.

Going back to the Register() method, we move the declaration of the resulting ServiceResponse to the top.

After that, we call our UserExists() method, and if the user does exist, we return the response with Success = false and a Message like User already exists..

public async Task<ServiceResponse<int>> Register(User user, string password)

{

ServiceResponse<int> response = new ServiceResponse<int>();

if (await UserExists(user.Username))

{

response.Success = false;

response.Message = "User already exists.";

return response;

}

CreatePasswordHash(password, out byte[] passwordHash, out byte[] passwordSalt);

user.PasswordHash = passwordHash;

user.PasswordSalt = passwordSalt;

await \_context.Users.AddAsync(user);

await \_context.SaveChangesAsync();

response.Data = user.Id;

return response;

}

So far, so good. To test this, we need a controller now. Let’s do that next.

## **Authentication Controller**

We add a new C# class AuthController in the Controllers folder, derive from ControllerBase, add the usual attributes and the Microsoft.AspNetCore.Mvc using directive.

[ApiController]

[Route("[controller]")]

public class AuthController : ControllerBase

{

}

From here we create a constructor and inject the IAuthRepository. We add dotnet\_rpg.Data to our class and initialize the field authRepo from the parameter. If you want, you can also change the name and add an underscore.

private readonly IAuthRepository \_authRepo;

public AuthController(IAuthRepository authRepository)

{

\_authRepo = authRepository;

}

Now, we add the Register() method. Starting with the signature of the method, we have to add the System.Threading.Tasks namespace and then we already see that we need a new DTO for the request. We could send the username and the password via the URL, but this is not recommended. It’s better to hide this information in the body.

So, let’s add a UserRegisterDto real quick. We create a folder User and then add a new C# class with two properties, the Username and the Password, both as string.

public class UserRegisterDto

{

public string Username { get; set; }

public string Password { get; set; }

}

Now we can use this DTO as a request object in the Register() method.

public async Task<IActionResult> Register(UserRegisterDto request)

{

}

We declare a ServiceResponse and call the Register() method of the \_authRepo. The idea behind the User parameter is, that we could add more information to the user upon registration, like an email address, birthdate, favorite color and so on. That’s why we use that object. Again, that’s totally up to you and you could rewrite the necessary methods to just send a username and a password without any additional information.

After the registration call, we check if the Success member of the response is false. If so, we return a BadRequest(), otherwise we return 200 OK.

One last thing is missing. We add the attribute [HttpPost] with the route ”Register” on top of the method.

[HttpPost(“Register”)]

public async Task<IActionResult> Register(UserRegisterDto request)

{

ServiceResponse<int> response = await \_authRepo.Register(

new User { Username = request.Username }, request.Password);

if(!response.Success) {

return BadRequest(response);

}

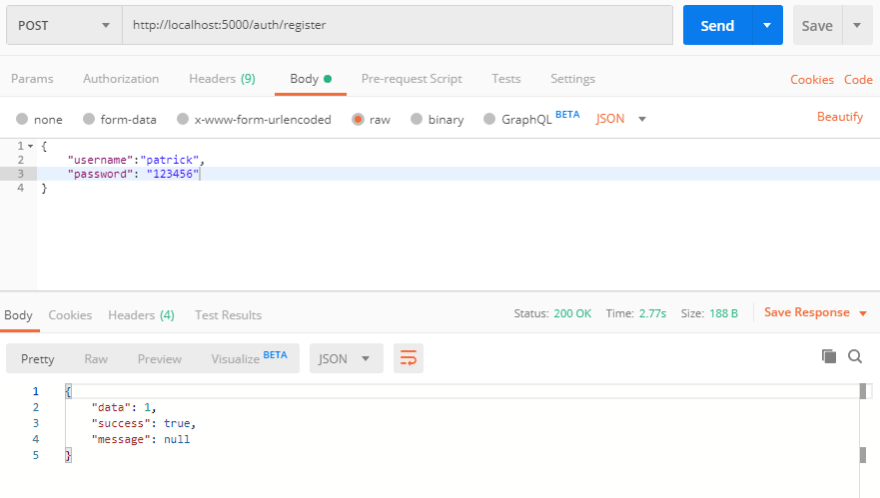
return Ok(response);

}

Finally, we have to register the AuthRepository in our Startup class. Similar to the CharacterService we can do that with AddScoped() and then the interface and the implementation class of the AuthRepository.

services.AddScoped<IAuthRepository, AuthRepository>();

Alright, start the Web API with dotnet watch run if it isn’t running and then let’s test the registration with Postman.

[](https://res.cloudinary.com/practicaldev/image/fetch/s---SweSM70--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/rwblnvm874kfnkj76055.png)

The HTTP method is POST, the URL is http://localhost:5000/auth/register and then we have to add a JSON body with a username and a password.

{

"username":"patrick",

"password": "123456"

}

The result is the usual ServiceResponse with the Id of the created User in the data field.

{

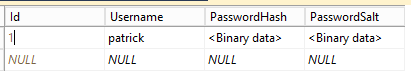
"data": 1,

"success": true,

"message": null

}

We can also find the new user in the database.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--Ee_t3ccx--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/9zytvclw8lqwbsv8nqhv.png)

Here we can also see that the id of the new user is 1 and that the password hash and the salt are binary data.

If we try to register with the exact same name again, we get the expected failing result back.

{

"data": 0,

"success": false,

"message": "User already exists."

}

Any kind of front end could read that now and display the message to the user.

Great! So, the account registration works. Let’s build the login next.

## **User Login**

Before we use any kind of token authentication, we will first implement the basic login and return the Id of the user as ServiceResponse.Data. Later, we will find the token here.

First, we need a method to verify the given password. Let’s call it VerifyPasswordHash() and return a boolean value to tell if the password is correct or not. The parameters are the password the user has entered, and the passwordHash as well as the passwordSalt of the user from the database.

With this information, we create a new instance of the HMACSHA512 class with the passwordSalt as key data. With that instance and the entered password we compute a new hash value.

This computedHash value will then be compared with the passwordHash from the database byte by byte. We use a simple for loop for that.

If the current byte value of the computedHash does not equal the passwordHash value with the same index, we know that the password is wrong. Otherwise, if the for loop went through with no unequal value, everything is fine and the user has been authenticated successfully.

private bool VerifyPasswordHash(string password, byte[] passwordHash, byte[] passwordSalt)

{

using (var hmac = new System.Security.Cryptography.HMACSHA512(passwordSalt))

{

var computedHash = hmac.ComputeHash(System.Text.Encoding.UTF8.GetBytes(password));

for (int i = 0; i < computedHash.Length; i++)

{

if (computedHash[i] != passwordHash[i])

{

return false;

}

}

return true;

}

}

Now we can build the login method and use the VerifyPasswordHash() method there.

First we add the async keyword and initialize a new ServiceResponse.

Then we have to find the requested user first. As usual, we do that by accessing the \_context and try to find the user with the given username.

If no user has been found, we send a failing ServiceResponse back and maybe a message as well.

Regarding the message, always take into account that you already reveal useful information to a potential attacker if you tell specifically that the user has not been found or that the password has been wrong. In these cases, an attacker would know that a particular user exists (it’s more critical with email addresses) and he or she could try to find the correct password with a brute force attack.

Anyways, since our application is just an example, let’s go with “User not found”. That way it’s easier for us to see if our web service works as expected.

Alright, we got the username covered, now we check if the password is correct. We finally call the VerifyPasswordHash() method with the entered password, and the PasswordHash and the PasswordSalt of the user from the database.

If the password is wrong, we send a failing ServiceResponse back again.

Finally, if the password is correct, we send the Id of the user back. As already mentioned, later we’ll send the token back here.

public async Task<ServiceResponse<string>> Login(string username, string password)

{

ServiceResponse<string> response = new ServiceResponse<string>();

User user = await \_context.Users.FirstOrDefaultAsync(x => x.Username.ToLower().Equals(username.ToLower()));

if (user == null)

{

response.Success = false;

response.Message = "User not found.";

}

else if (!VerifyPasswordHash(password, user.PasswordHash, user.PasswordSalt))

{

response.Success = false;

response.Message = "Wrong password.";

}

else

{

response.Data = user.Id.ToString();

}

return response;

}

Now we have to call this Login() method with the help of the AuthController.

But first, we create a new UserLoginDto with two properties, the Username and the Password. I know, it’s exactly the same as the UserRegisterDto. But again, maybe you want to add more information to the registration later, so if you split the DTOs now, you don’t have to do it later.

Back to the AuthController we add the Login() method. We can actually copy the Register() method and just make some small changes.

We change the route, the method name and the parameter type first.

The ServiceResponse type is now a string, we call the Login() method of the \_authRepo and give that method the request.Username and the request.Password.

[HttpPost("Login")]

public async Task<IActionResult> Login(UserLoginDto request)

{

ServiceResponse<string> response = await \_authRepo.Login(

request.Username, request.Password);

if(!response.Success) {

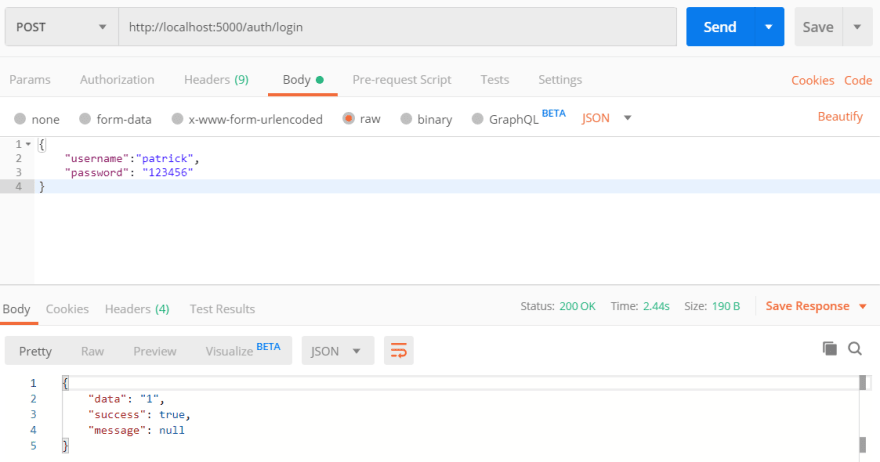
return BadRequest(response);

}

return Ok(response);

}

That’s it already. Let’s test that now in Postman. We should still have the user available from the registration. So we can use the same JSON body, the HTTP method is still POST, but the route is now login.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--z2aBUZEq--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/kef649r0tu0txe6zvx54.png)

Great. We get the correct id back.

What happens if the username is wrong?

{

"data": null,

"success": false,

"message": "User not found."

}

The user has not been found. What about a wrong password?

{

"data": null,

"success": false,

"message": "Wrong password."

}

We get the information that the password is indeed wrong.

Perfect.

It’s time to move on with token authentication.

That's it for the 7th part of this tutorial series. I hope it was useful for you. To get notified for the next part, simply follow me here on [dev.to](https://dev.to/_patrickgod) or [subscribe to my newsletter](https://mailchi.mp/96620175570f/dotnetcore). You'll be the first to know.

See you next time!

Take care.

Next up: Authentication with JSON Web Tokens in .NET Core

*Image created by cornecoba on*[*freepik.com*](https://www.freepik.com/free-vector/flying-rocket-background_1130797.htm)*.*

### But wait, there’s more!

* Let’s connect on [Twitter](https://twitter.com/_PatrickGod), [YouTube](https://www.youtube.com/channel/UCq8LldVrjqe61KQttZlLW8g), [LinkedIn](https://www.linkedin.com/in/patrickgod) or here on [dev.to](https://dev.to/_patrickgod).
* Get the [5 Software Developer’s Career Hacks](http://patrickgod.com/5-software-developers-career-hacks/) for free.
* Enjoy more valuable articles for your developer life and career on [patrickgod.com](http://patrickgod.com/).