
Otp.NET

An implementation TOTP RFC 6238 and HOTP RFC 4226 in C#.



Get it on NuGet

<https://www.nuget.org/packages/Otp.NET>

```
1 Install-Package Otp.NET
```

or

```
1 dotnet add package Otp.NET
```

Documentation

- TOTP (Timed One Time Password)
- HOTP (HMAC-based One Time Password)
- OTP Uri
- Base32 Encoding

TOTP (Timed One Time Password)

TOTP is an algorithm that uses a rolling window of time to calculate single use passwords. It is often used for two factor authentication. The Google Authenticator app uses TOTP to calculate one time passwords. This library implements TOTP code calculation in C#. This could be embedded in a mobile app using Mono, or used server side to simply validate codes that are provided.

Creation of a TOTP object Use of the library is fairly straightforward. There is a class called Totp. Simply create a new instance of it and pass in the shared secret key in plaintext as a byte array.

```
1 using OtpNet;
```

```
1 var totp = new Totp(secretKey);
```

There are several options that can be used to change how the code is calculated. These are all mentioned in the RFC. These options are specified when the TOTP object is created.

Different hash algorithms can be used to calculate the code. The default is Sha1, but Sha256, and Sha512 may be used instead.

To change that behavior from the default of Sha1 simply pass in the OtpHashMode enum with the desired value into the constructor.

```
1 var totp = new Totp(secretKey, mode: OtpHashMode.Sha512);
```

The time step window can also be specified. The RFC recommends a window of thirty seconds. That means that a new code will be generated every thirty seconds. The step window can be changed however if required. There are not tests around this as the RFC test tables all use a 30 second window so use this feature at your own risk. Like the hash mode, pass this value into the constructor. It is an int that represents seconds.

```
1 var totp = new Totp(secretKey, step: 15); // a new code will be
    generated every 15 seconds
```

Finally the truncation level can be specified. Basically this is how many digits do you want your TOTP code to be. The tests in the RFC specify 8, but 6 has become a de-facto standard if not an actual one. For this reason the default is 6 but you can set it to something else. There aren't a lot of tests around this either so use at your own risk (other than the fact that the RFC test table uses TOTP values that are 8 digits).

```
1 var totp = new Totp(secretKey, totpSize: 8);
```

Code Calculation Once you have an instance of the Totp class, you can easily calculate a code by Calling the ComputeTotp method. You need to provide the timestamp to use in the code calculation. DateTime.UtcNow is the recommended value. There is an overload that doesn't take a parameter that just uses UtcNow.

```
1 var totpCode = totp.ComputeTotp(DateTime.UtcNow);
2 // or use the overload that uses UtcNow
3 var totpCode = totp.ComputeTotp();
```

Remaining Time There is a method that will tell you how much time remains in the current time step window in seconds.

```
1 var remainingTime = totp.RemainingSeconds();
2 // there is also an overload that lets you specify the time
3 var remainingSeconds = totp.RemainingSeconds(DateTime.UtcNow);
```

Verification The TOTP implementation provides a mechanism for verifying TOTP codes that are passed in. There is a method called `VerifyTotp` with an overload that takes a specific timestamp.

```
1 public bool VerifyTotp(string totp, out long timeWindowUsed,
   VerificationWindow window = null);
2 public bool VerifyTotp(DateTime timestamp, string totp, out long
   timeWindowUsed, VerificationWindow window = null)
```

If the overload that doesn't take a timestamp is called, `DateTime.UtcNow` will be used as the operand.

One Time Use There is an output long called `timeWindowUsed`. This is provided so that the caller of the function can persist/check that the code has only been validated once. RFC 6238 Section 5.2 states that a code must only be accepted once. The output parameter reports the specific time window where the match occurred for persistence comparison in future verification attempts.

It is up to the consumer of this library to ensure that only one match for a given time step window is actually accepted. This library will only go so far as to determine that there was a valid code provided given the current time and the key, not that it was truly used one time as this library has no persistence.

Expanded time Window RFC 6238 Section 5.2 defines the recommended conditions for accepting a TOTP validation code. The exact text in the RFC is "We RECOMMEND that at most one time step is allowed as the network delay."

The `VerifyTotp` method takes an optional `VerificationWindow` parameter. This parameter allows you to define the window of steps that are considered acceptable. The actual step where the match was found will be reported in the aforementioned output parameter.

The default is that no delay will be accepted and the code must match the current code in order to be considered a match. Simply omitting the optional parameter will cause this default behavior.

If a time delay is required, a `VerificationWindow` object can be provided that describes the acceptable range of values to check.

To allow a delay as per the recommendation of the RFC (one time step delay) create a verification window object as follows

```
1 var window = new VerificationWindow(previous:1, future:1);
```

This means that the current step, and 1 step prior to the current will be allowed in the match. If you wanted to accept 5 steps backward (not recommended in the RFC) then you would change the `previous` parameter to 5.

There is also a parameter called `future` that allows you to match future codes. This might be useful if the client is ahead of the server by just enough that the code provided is slightly ahead.

Ideally the times should match and every effort should be taken to ensure that the client and server times are in sync.

It is not recommended to provide any value other than the default (current frame only) or the RFC recommendation of this and one frame prior as well as one frame ahead (see below)

```
1 var window = new VerificationWindow(previous:1, future:1);
```

In order to make using the RFC recommendation easier, there is a constant that contains a verification window object that complies with the RFC recommendation.

```
1 VerificationWindow.RfcSpecifiedNetworkDelay
```

This can be used as follows

```
1 totp.VerifyTotp(totpCode, out timeWindowUsed, VerificationWindow.  
    RfcSpecifiedNetworkDelay);
```

Time compensation In an ideal world both the client and the server's system time are correct to the second with NIST or other authoritative time standards. This would ensure that the generated code is always correct. If at all possible, sync the system time as closely as with NIST.

There are cases where this simply isn't possible. Perhaps you are writing an app to generate codes for use with a server who's time is significantly off. You can't control the erroneous time of the server. You could set your system clock to match but then your time would be off significantly which isn't the desired result. There is a class called `TimeCorrection` that helps with these cases.

A time correction object creates an offset that can be used to correct (at least for the purposes of this calculation) the time relative to the incorrect system time.

It is created as follows

```
1 var correction = new TimeCorrection(correctTime);
```

Where the correct time parameter is a `DateTime` object that represents the current correct (at least for the purposes of verification) UTC time. For this to work there needs to be some way to get an instance of the current acceptable time. This could be done with an NTP (NTP with NIST is coming soon in this library) or looking for a Date response header from an HTTP request or some other way.

Once this instance is created it can be used for a long time as it always will use the current system time as a base to apply the correction factor. The object is threadsafe and thus can be used by multiple threads or web requests simultaneously.

There is an overload that takes both the correct time and the reference time to use as well. This can be used in cases where UTC time isn't used.

The Totp class constructor can take a TimeCorrection object that will be applied to all time calculations and verifications.

```
1 var totp = new Totp(secretKey, timeCorrection: correction);
```

HOTP (HMAC-based One Time Password)

In addition to TOTP, this library implements HOTP (counter based) code calculation in C#.

Creation of an HOTP object

```
1 using OtpNet;
```

```
1 var hotp = new Hotp(secretKey);
```

There are several options that can be used to change how the code is calculated. These are all mentioned in the RFC. These options are specified when the HOTP object is created.

Different hash algorithms can be used to calculate the code. The default is Sha1, but Sha256, and Sha512 may be used instead.

To change that behavior from the default of Sha1 simply pass in the OtpHashMode enum with the desired value into the constructor.

```
1 var hotp = new Hotp(secretKey, mode: OtpHashMode.Sha512);
```

Finally the truncation level can be specified. Basically this is how many digits do you want your HOTP code to be. The tests in the RFC specify 8, but 6 has become a de-facto standard if not an actual one. For this reason the default is 6 but you can set it to something else. There aren't a lot of tests around this either so use at your own risk (other than the fact that the RFC test table uses HOTP values that are 8 digits).

```
1 var hotp = new Hotp(secretKey, hotpSize: 8);
```

Verification The HOTP implementation provides a mechanism for verifying HOTP codes that are passed in. There is a method called VerifyHotp with an overload that takes a counter value.

```
1 public bool VerifyHotp(string totp, long counter);
```

OTP Uri

You can use the `OtpUri` class to generate OTP style uris in the “Key Uri Format” as defined here: <https://github.com/google/google-authenticator/wiki/Key-Uri-Format>

```
1 var uriString = new OtpUri(OtpType.Totp, "JBSWY3DPEHPK3PXP", "  
    alice@google.com", "ACME Co").ToString();  
2 // uriString is otpauth://totp/ACME%20Co:alice%40google.com?secret=  
    JBSWY3DPEHPK3PXP&issuer=ACME%20Co&algorithm=SHA1&digits=6&period=30
```

Base32 Encoding

Also included is a Base32 helper.

```
1 var key = KeyGeneration.GenerateRandomKey(20);  
2  
3 var base32String = Base32Encoding.ToString(key);  
4 var base32Bytes = Base32Encoding.ToBytes(base32String);  
5  
6 var otp = new Totp(base32Bytes);
```

Credits

This project is originally based on the `OtpSharp` library. `OtpSharp` was written by Devin Martin.