Packages:

1) Back-End Packages:

1.1) Mongoose:

**Getting Started**

*First be sure you have*[*MongoDB*](http://www.mongodb.org/downloads)*and*[*Node.js*](http://nodejs.org/)*installed.*

Next install Mongoose from the command line using npm:

$ npm install mongoose

Now say we like fuzzy kittens and want to record every kitten we ever meet in MongoDB. The first thing we need to do is include mongoose in our project and open a connection to the test database on our locally running instance of MongoDB.

*// getting-started.js*

**var** mongoose = require('mongoose');

mongoose.connect('mongodb://localhost/test', {useNewUrlParser: true});

We have a pending connection to the test database running on localhost. We now need to get notified if we connect successfully or if a connection error occurs:

**var** db = mongoose.connection;

db.on('error', console.error.bind(console, 'connection error:'));

db.once('open', **function**() {

*// we're connected!*

});

Once our connection opens, our callback will be called. For brevity, let's assume that all following code is within this callback.

With Mongoose, everything is derived from a [Schema](https://mongoosejs.com/docs/guide.html). Let's get a reference to it and define our kittens.

**var** kittySchema = **new** mongoose.Schema({

name: String

});

So far so good. We've got a schema with one property, name, which will be a String. The next step is compiling our schema into a [Model](https://mongoosejs.com/docs/models.html).

**var** Kitten = mongoose.model('Kitten', kittySchema);

A model is a class with which we construct documents. In this case, each document will be a kitten with properties and behaviors as declared in our schema. Let's create a kitten document representing the little guy we just met on the sidewalk outside:

**var** silence = **new** Kitten({ name: 'Silence' });

console.log(silence.name); *// 'Silence'*

Kittens can meow, so let's take a look at how to add "speak" functionality to our documents:

*// NOTE: methods must be added to the schema before compiling it with mongoose.model()*

kittySchema.methods.speak = **function** () {

**var** greeting = **this**.name

? "Meow name is " + **this**.name

: "I don't have a name";

console.log(greeting);

}

**var** Kitten = mongoose.model('Kitten', kittySchema);

Functions added to the methods property of a schema get compiled into the Model prototype and exposed on each document instance:

**var** fluffy = **new** Kitten({ name: 'fluffy' });

fluffy.speak(); *// "Meow name is fluffy"*

We have talking kittens! But we still haven't saved anything to MongoDB. Each document can be saved to the database by calling its [save](https://mongoosejs.com/docs/api.html#model_Model-save) method. The first argument to the callback will be an error if any occurred.

fluffy.save(**function** (err, fluffy) {

**if** (err) **return** console.error(err);

fluffy.speak();

});

Say time goes by and we want to display all the kittens we've seen. We can access all of the kitten documents through our Kitten [model](https://mongoosejs.com/docs/models.html).

Kitten.find(**function** (err, kittens) {

**if** (err) **return** console.error(err);

console.log(kittens);

})

We just logged all of the kittens in our db to the console. If we want to filter our kittens by name, Mongoose supports MongoDBs rich [querying](https://mongoosejs.com/docs/queries.html) syntax.

Kitten.find({ name: /^fluff/ }, callback);

This performs a search for all documents with a name property that begins with "Fluff" and returns the result as an array of kittens to the callback.

**Congratulations**

That's the end of our quick start. We created a schema, added a custom document method, saved and queried kittens in MongoDB using Mongoose. Head over to the [guide](https://mongoosejs.com/docs/guide.html), or [API docs](https://mongoosejs.com/docs/api.html) for more.

1.2) bcryptjs

**Usage**

The library is compatible with CommonJS and AMD loaders and is exposed globally as dcodeIO.bcrypt if neither is available.

**node.js**

On node.js, the inbuilt [crypto module](http://nodejs.org/api/crypto.html)'s randomBytes interface is used to obtain secure random numbers.

npm install bcryptjs

var bcrypt **=** require('bcryptjs');

**...**

**Browser**

In the browser, bcrypt.js relies on [Web Crypto API](http://www.w3.org/TR/WebCryptoAPI)'s getRandomValues interface to obtain secure random numbers. If no cryptographically secure source of randomness is available, you may specify one through [bcrypt.setRandomFallback](https://github.com/dcodeIO/bcrypt.js#setrandomfallbackrandom).

var bcrypt **=** dcodeIO.bcrypt;

**...**

or

require.config({

    paths**:** { "bcrypt"**:** "/path/to/bcrypt.js" }

});

require(["bcrypt"], function(bcrypt) {

**...**

});

**Usage - Sync**

To hash a password:

var bcrypt **=** require('bcryptjs');

var salt **=** bcrypt.genSaltSync(10);

var hash **=** bcrypt.hashSync("B4c0/\/", salt);

*// Store hash in your password DB.*

To check a password:

*// Load hash from your password DB.*

bcrypt.compareSync("B4c0/\/", hash); *// true*

bcrypt.compareSync("not\_bacon", hash); *// false*

Auto-gen a salt and hash:

var hash **=** bcrypt.hashSync('bacon', 8);

**Usage - Async**

To hash a password:

var bcrypt **=** require('bcryptjs');

bcrypt.genSalt(10, function(err, salt) {

    bcrypt.hash("B4c0/\/", salt, function(err, hash) {

*// Store hash in your password DB.*

    });

});

To check a password:

*// Load hash from your password DB.*

bcrypt.compare("B4c0/\/", hash, function(err, res) {

*// res === true*

});

bcrypt.compare("not\_bacon", hash, function(err, res) {

*// res === false*

});

*// As of bcryptjs 2.4.0, compare returns a promise if callback is omitted:*

bcrypt.compare("B4c0/\/", hash).then((res) => {

*// res === true*

});

Auto-gen a salt and hash:

bcrypt.hash('bacon', 8, function(err, hash) {

});

**Note:** Under the hood, asynchronisation splits a crypto operation into small chunks. After the completion of a chunk, the execution of the next chunk is placed on the back of [JS event loop queue](https://developer.mozilla.org/en/docs/Web/JavaScript/EventLoop), thus efficiently sharing the computational resources with the other operations in the queue.

**API**

**setRandomFallback(random)**

Sets the pseudo random number generator to use as a fallback if neither node's cryptomodule nor the Web Crypto API is available. Please note: It is highly important that the PRNG used is cryptographically secure and that it is seeded properly!

| **Parameter** | **Type** | **Description** |
| --- | --- | --- |
| random | *function(number):!Array.<number>* | Function taking the number of bytes to generate as its sole argument, returning the corresponding array of cryptographically secure random byte values. |
| **@see** |  | <http://nodejs.org/api/crypto.html> |
| **@see** |  | <http://www.w3.org/TR/WebCryptoAPI/> |

**Hint:** You might use [isaac.js](https://github.com/rubycon/isaac.js) as a CSPRNG but you still have to make sure to seed it properly.

**genSaltSync(rounds=, seed\_length=)**

Synchronously generates a salt.

| **Parameter** | **Type** | **Description** |
| --- | --- | --- |
| rounds | *number* | Number of rounds to use, defaults to 10 if omitted |
| seed\_length | *number* | Not supported. |
| **@returns** | *string* | Resulting salt |
| **@throws** | *Error* | If a random fallback is required but not set |

**genSalt(rounds=, seed\_length=, callback)**

Asynchronously generates a salt.

| **Parameter** | **Type** | **Description** |
| --- | --- | --- |
| rounds | *number | function(Error, string=)* | Number of rounds to use, defaults to 10 if omitted |
| seed\_length | *number | function(Error, string=)* | Not supported. |
| callback | *function(Error, string=)* | Callback receiving the error, if any, and the resulting salt |
| **@returns** | *Promise* | If callback has been omitted |
| **@throws** | *Error* | If callback is present but not a function |

**hashSync(s, salt=)**

Synchronously generates a hash for the given string.

| **Parameter** | **Type** | **Description** |
| --- | --- | --- |
| s | *string* | String to hash |
| salt | *number | string* | Salt length to generate or salt to use, default to 10 |
| **@returns** | *string* | Resulting hash |

**hash(s, salt, callback, progressCallback=)**

Asynchronously generates a hash for the given string.

| **Parameter** | **Type** | **Description** |
| --- | --- | --- |
| s | *string* | String to hash |
| salt | *number | string* | Salt length to generate or salt to use |
| callback | *function(Error, string=)* | Callback receiving the error, if any, and the resulting hash |
| progressCallback | *function(number)* | Callback successively called with the percentage of rounds completed (0.0 - 1.0), maximally once per MAX\_EXECUTION\_TIME = 100 ms. |
| **@returns** | *Promise* | If callback has been omitted |
| **@throws** | *Error* | If callback is present but not a function |

**compareSync(s, hash)**

Synchronously tests a string against a hash.

| **Parameter** | **Type** | **Description** |
| --- | --- | --- |
| s | *string* | String to compare |
| hash | *string* | Hash to test against |
| **@returns** | *boolean* | true if matching, otherwise false |
| **@throws** | *Error* | If an argument is illegal |

**compare(s, hash, callback, progressCallback=)**

Asynchronously compares the given data against the given hash.

| **Parameter** | **Type** | **Description** |
| --- | --- | --- |
| s | *string* | Data to compare |
| hash | *string* | Data to be compared to |
| callback | *function(Error, boolean)* | Callback receiving the error, if any, otherwise the result |
| progressCallback | *function(number)* | Callback successively called with the percentage of rounds completed (0.0 - 1.0), maximally once per MAX\_EXECUTION\_TIME = 100 ms. |
| **@returns** | *Promise* | If callback has been omitted |
| **@throws** | *Error* | If callback is present but not a function |

**getRounds(hash)**

Gets the number of rounds used to encrypt the specified hash.

| **Parameter** | **Type** | **Description** |
| --- | --- | --- |
| hash | *string* | Hash to extract the used number of rounds from |
| **@returns** | *number* | Number of rounds used |
| **@throws** | *Error* | If hash is not a string |

**getSalt(hash)**

Gets the salt portion from a hash. Does not validate the hash.

| **Parameter** | **Type** | **Description** |
| --- | --- | --- |
| hash | *string* | Hash to extract the salt from |
| **@returns** | *string* | Extracted salt part |
| **@throws** | *Error* | If hash is not a string or otherwise invalid |

1.3) passport

# Overview

Passport is authentication middleware for [Node](http://nodejs.org/). It is designed to serve a singular purpose: authenticate requests. When writing modules, encapsulation is a virtue, so Passport delegates all other functionality to the application. This separation of concerns keeps code clean and maintainable, and makes Passport extremely easy to integrate into an application.

In modern web applications, authentication can take a variety of forms. Traditionally, users log in by providing a username and password. With the rise of social networking, single sign-on using an [OAuth](http://oauth.net/) provider such as [Facebook](https://www.facebook.com/) or [Twitter](https://twitter.com/) has become a popular authentication method. Services that expose an API often require token-based credentials to protect access.

Passport recognizes that each application has unique authentication requirements. Authentication mechanisms, known as strategies, are packaged as individual modules. Applications can choose which strategies to employ, without creating unnecessary dependencies.

Despite the complexities involved in authentication, code does not have to be complicated.

app.post('/login', passport.authenticate('local', { successRedirect: '/',

failureRedirect: '/login' }));

## Install

$ npm install passport

# Authenticate

Authenticating requests is as simple as calling passport.authenticate() and specifying which strategy to employ. authenticate()'s function signature is standard [Connect](http://www.senchalabs.org/connect/) middleware, which makes it convenient to use as route middleware in [Express](http://expressjs.com/) applications.

app.post('/login',

passport.authenticate('local'),

**function**(req, res) {

// If this function gets called, authentication was successful.

// `req.user` contains the authenticated user.

res.redirect('/users/' + req.user.username);

});

By default, if authentication fails, Passport will respond with a 401 Unauthorized status, and any additional route handlers will not be invoked. If authentication succeeds, the next handler will be invoked and the req.user property will be set to the authenticated user.

Note: Strategies must be configured prior to using them in a route. Continue reading the chapter on [configuration](http://www.passportjs.org/guide/configure/) for details.

## Redirects

A redirect is commonly issued after authenticating a request.

app.post('/login',

passport.authenticate('local', { successRedirect: '/',

failureRedirect: '/login' }));

In this case, the redirect options override the default behavior. Upon successful authentication, the user will be redirected to the home page. If authentication fails, the user will be redirected back to the login page for another attempt.

## Flash Messages

Redirects are often combined with flash messages in order to display status information to the user.

app.post('/login',

passport.authenticate('local', { successRedirect: '/',

failureRedirect: '/login',

failureFlash: true })

);

Setting the failureFlash option to true instructs Passport to flash an error message using the message given by the strategy's verify callback, if any. This is often the best approach, because the verify callback can make the most accurate determination of why authentication failed.

Alternatively, the flash message can be set specifically.

passport.authenticate('local', { failureFlash: 'Invalid username or password.' });

A successFlash option is available which flashes a success message when authentication succeeds.

passport.authenticate('local', { successFlash: 'Welcome!' });

Note: Using flash messages requires a req.flash() function. Express 2.x provided this functionality, however it was removed from Express 3.x. Use of [connect-flash](https://github.com/jaredhanson/connect-flash) middleware is recommended to provide this functionality when using Express 3.x.

## Disable Sessions

After successful authentication, Passport will establish a persistent login session. This is useful for the common scenario of users accessing a web application via a browser. However, in some cases, session support is not necessary. For example, API servers typically require credentials to be supplied with each request. When this is the case, session support can be safely disabled by setting the session option to false.

app.get('/api/users/me',

passport.authenticate('basic', { session: false }),

**function**(req, res) {

res.json({ id: req.user.id, username: req.user.username });

});

## Custom Callback

If the built-in options are not sufficient for handling an authentication request, a custom callback can be provided to allow the application to handle success or failure.

app.get('/login', **function**(req, res, next) {

passport.authenticate('local', **function**(err, user, info) {

**if** (err) { **return** next(err); }

**if** (!user) { **return** res.redirect('/login'); }

req.logIn(user, **function**(err) {

**if** (err) { **return** next(err); }

**return** res.redirect('/users/' + user.username);

});

})(req, res, next);

});

In this example, note that authenticate() is called from within the route handler, rather than being used as route middleware. This gives the callback access to the req and res objects through closure.

If authentication failed, user will be set to false. If an exception occurred, err will be set. An optional info argument will be passed, containing additional details provided by the strategy's verify callback.

The callback can use the arguments supplied to handle the authentication result as desired. Note that when using a custom callback, it becomes the application's responsibility to establish a session (by calling req.login()) and send a response.

# Configure

Three pieces need to be configured to use Passport for authentication:

1. Authentication strategies
2. Application middleware
3. Sessions (optional)

## Strategies

Passport uses what are termed strategies to authenticate requests. Strategies range from verifying a username and password, delegated authentication using [OAuth](http://oauth.net/) or federated authentication using [OpenID](http://openid.net/).

Before asking Passport to authenticate a request, the strategy (or strategies) used by an application must be configured.

Strategies, and their configuration, are supplied via the use() function. For example, the following uses the LocalStrategy for username/password authentication.

**var** passport = **require**('passport')

, LocalStrategy = **require**('passport-local').Strategy;

passport.use(**new** LocalStrategy(

**function**(username, password, done) {

User.findOne({ username: username }, **function** (err, user) {

**if** (err) { **return** done(err); }

**if** (!user) {

**return** done(null, false, { message: 'Incorrect username.' });

}

**if** (!user.validPassword(password)) {

**return** done(null, false, { message: 'Incorrect password.' });

}

**return** done(null, user);

});

}

));

## Verify Callback

This example introduces an important concept. Strategies require what is known as a verify callback. The purpose of a verify callback is to find the user that possesses a set of credentials.

When Passport authenticates a request, it parses the credentials contained in the request. It then invokes the verify callback with those credentials as arguments, in this case username and password. If the credentials are valid, the verify callback invokes done to supply Passport with the user that authenticated.

**return** done(null, user);

If the credentials are not valid (for example, if the password is incorrect), done should be invoked with false instead of a user to indicate an authentication failure.

**return** done(null, false);

An additional info message can be supplied to indicate the reason for the failure. This is useful for displaying a flash message prompting the user to try again.

**return** done(null, false, { message: 'Incorrect password.' });

Finally, if an exception occurred while verifying the credentials (for example, if the database is not available), done should be invoked with an error, in conventional Node style.

**return** done(err);

Note that it is important to distinguish the two failure cases that can occur. The latter is a server exception, in which err is set to a non-null value. Authentication failures are natural conditions, in which the server is operating normally. Ensure that err remains null, and use the final argument to pass additional details.

By delegating in this manner, the verify callback keeps Passport database agnostic. Applications are free to choose how user information is stored, without any assumptions imposed by the authentication layer.

#### Middleware

In a [Connect](http://senchalabs.github.com/connect/) or [Express](http://expressjs.com/)-based application, passport.initialize() middleware is required to initialize Passport. If your application uses persistent login sessions, passport.session() middleware must also be used.

app.configure(**function**() {

app.use(express.static('public'));

app.use(express.cookieParser());

app.use(express.bodyParser());

app.use(express.session({ secret: 'keyboard cat' }));

app.use(passport.initialize());

app.use(passport.session());

app.use(app.router);

});

Note that enabling session support is entirely optional, though it is recommended for most applications. If enabled, be sure to use session() before passport.session() to ensure that the login session is restored in the correct order.

In Express 4.x, the Connect middleware is no longer included in the Express core, and the app.configure() method has been removed. The same middleware can be found in their npm module equivalents.

**var** session = **require**("express-session"),

bodyParser = **require**("body-parser");

app.use(express.static("public"));

app.use(session({ secret: "cats" }));

app.use(bodyParser.urlencoded({ extended: false }));

app.use(passport.initialize());

app.use(passport.session());

#### Sessions

In a typical web application, the credentials used to authenticate a user will only be transmitted during the login request. If authentication succeeds, a session will be established and maintained via a cookie set in the user's browser.

Each subsequent request will not contain credentials, but rather the unique cookie that identifies the session. In order to support login sessions, Passport will serialize and deserialize user instances to and from the session.

passport.serializeUser(**function**(user, done) {

done(null, user.id);

});

passport.deserializeUser(**function**(id, done) {

User.findById(id, **function**(err, user) {

done(err, user);

});

});

In this example, only the user ID is serialized to the session, keeping the amount of data stored within the session small. When subsequent requests are received, this ID is used to find the user, which will be restored to req.user.

The serialization and deserialization logic is supplied by the application, allowing the application to choose an appropriate database and/or object mapper, without imposition by the authentication layer.

# Username & Password

The most widely used way for websites to authenticate users is via a username and password. Support for this mechanism is provided by the [passport-local](https://github.com/jaredhanson/passport-local) module.

## Install

$ npm install passport-local

## Configuration

**var** passport = **require**('passport')

, LocalStrategy = **require**('passport-local').Strategy;

passport.use(**new** LocalStrategy(

**function**(username, password, done) {

User.findOne({ username: username }, **function**(err, user) {

**if** (err) { **return** done(err); }

**if** (!user) {

**return** done(null, false, { message: 'Incorrect username.' });

}

**if** (!user.validPassword(password)) {

**return** done(null, false, { message: 'Incorrect password.' });

}

**return** done(null, user);

});

}

));

The verify callback for local authentication accepts username and password arguments, which are submitted to the application via a login form.

## Form

A form is placed on a web page, allowing the user to enter their credentials and log in.

<form action="/login" method="post">

<div>

<label>Username:</label>

<input type="text" name="username"/>

</div>

<div>

<label>Password:</label>

<input type="password" name="password"/>

</div>

<div>

<input type="submit" value="Log In"/>

</div>

</form>

## Route

The login form is submitted to the server via the POST method. Using authenticate() with the localstrategy will handle the login request.

app.post('/login',

passport.authenticate('local', { successRedirect: '/',

failureRedirect: '/login',

failureFlash: true })

);

Setting the failureFlash option to true instructs Passport to flash an error message using the messageoption set by the verify callback above. This is helpful when prompting the user to try again.

## Parameters

By default, LocalStrategy expects to find credentials in parameters named username and password. If your site prefers to name these fields differently, options are available to change the defaults.

passport.**use**(**new** LocalStrategy({

usernameField: 'email',

passwordField: 'passwd'

},

**function**(username, password, done) {

// ...

}

));

# OpenID

[OpenID](http://openid.net/) is an open standard for federated authentication. When visiting a website, users present their OpenID to sign in. The user then authenticates with their chosen OpenID provider, which issues an assertion to confirm the user's identity. The website verifies this assertion in order to sign the user in.

Support for OpenID is provided by the [passport-openid](https://github.com/jaredhanson/passport-openid) module.

## Install

$ npm install passport-openid

## Configuration

When using OpenID, a return URL and realm must be specified. The returnURL is the URL to which the user will be redirected after authenticating with their OpenID provider. realm indicates the part of URL-space for which authentication is valid. Typically this will be the root URL of the website.

**var** passport = **require**('passport')

, OpenIDStrategy = **require**('passport-openid').Strategy;

passport.use(**new** OpenIDStrategy({

returnURL: 'http://www.example.com/auth/openid/return',

realm: 'http://www.example.com/'

},

**function**(identifier, done) {

User.findOrCreate({ openId: identifier }, **function**(err, user) {

done(err, user);

});

}

));

The verify callback for OpenID authentication accepts an identifier argument containing the user's claimed identifier.

## Form

A form is placed on a web page, allowing the user to enter their OpenID and sign in.

<form action="/auth/openid" method="post">

<div>

<label>OpenID:</label>

<input type="text" name="openid\_identifier"/><br/>

</div>

<div>

<input type="submit" value="Sign In"/>

</div>

</form>

## Routes

Two routes are required for OpenID authentication. The first route accepts the form submission containing an OpenID identifier. During authentication, the user will be redirected to their OpenID provider. The second route is the URL to which the user will be returned after authenticating with their OpenID provider.

// Accept the OpenID identifier and redirect the user to their OpenID

// provider for authentication. When complete, the provider will redirect

// the user back to the application at:

// /auth/openid/return

app.post('/auth/openid', passport.authenticate('openid'));

// The OpenID provider has redirected the user back to the application.

// Finish the authentication process by verifying the assertion. If valid,

// the user will be logged in. Otherwise, authentication has failed.

app.get('/auth/openid/return',

passport.authenticate('openid', { successRedirect: '/',

failureRedirect: '/login' }));

## Profile Exchange

OpenID can optionally be configured to retrieve profile information about the user being authenticated. Profile exchange is enabled by setting the profile option to true.

passport.use(**new** OpenIDStrategy({

returnURL: 'http://www.example.com/auth/openid/return',

realm: 'http://www.example.com/',

profile: true

},

**function**(identifier, profile, done) {

// ...

}

));

When profile exchange is enabled, the function signature of the verify callback accepts an additional profile argument containing user profile information provided by the OpenID provider; refer to [User Profile](http://www.passportjs.org/guide/profile/) for further information.

# OAuth

[OAuth](http://oauth.net/) is a standard protocol that allows users to authorize API access to web and desktop or mobile applications. Once access has been granted, the authorized application can utilize the API on behalf of the user. OAuth has also emerged as a popular mechanism for [delegated authentication](http://hueniverse.com/2009/04/introducing-sign-in-with-twitter-oauth-style-connect/).

OAuth comes in two primary flavors, both of which are widely deployed.

The initial version of OAuth was developed as an open standard by a loosely organized collective of web developers. Their work resulted in [OAuth 1.0](http://oauth.net/core/1.0/), which was superseded by [OAuth 1.0a](http://oauth.net/core/1.0a/). This work has now been standardized by the [IETF](http://www.ietf.org/) as [RFC 5849](http://tools.ietf.org/html/rfc5849).

Recent efforts undertaken by the [Web Authorization Protocol Working Group](http://tools.ietf.org/wg/oauth/) have focused on defining [OAuth 2.0](http://tools.ietf.org/html/rfc6749). Due to the lengthy standardization effort, providers have proceeded to deploy implementations conforming to various drafts, each with slightly different semantics.

Thankfully, Passport shields an application from the complexities of dealing with OAuth variants. In many cases, a provider-specific strategy can be used instead of the generic OAuth strategies described below. This cuts down on the necessary configuration, and accommodates any provider-specific quirks. See [Facebook](http://www.passportjs.org/docs#facebook), [Twitter](http://www.passportjs.org/docs#twitter) or the list of [providers](http://www.passportjs.org/docs/downloads/html/) for preferred usage.

Support for OAuth is provided by the [passport-oauth](https://github.com/jaredhanson/passport-oauth) module.

## Install

$ npm install passport-oauth

### OAuth 1.0

OAuth 1.0 is a delegated authentication strategy that involves multiple steps. First, a request token must be obtained. Next, the user is redirected to the service provider to authorize access. Finally, after authorization has been granted, the user is redirected back to the application and the request token can be exchanged for an access token. The application requesting access, known as a consumer, is identified by a consumer key and consumer secret.

## Configuration

When using the generic OAuth strategy, the key, secret, and endpoints are specified as options.

**var** passport = **require**('passport')

, OAuthStrategy = **require**('passport-oauth').OAuthStrategy;

passport.use('provider', **new** OAuthStrategy({

requestTokenURL: 'https://www.provider.com/oauth/request\_token',

accessTokenURL: 'https://www.provider.com/oauth/access\_token',

userAuthorizationURL: 'https://www.provider.com/oauth/authorize',

consumerKey: '123-456-789',

consumerSecret: 'shhh-its-a-secret'

callbackURL: 'https://www.example.com/auth/provider/callback'

},

**function**(token, tokenSecret, profile, done) {

User.findOrCreate(..., **function**(err, user) {

done(err, user);

});

}

));

The verify callback for OAuth-based strategies accepts token, tokenSecret, and profile arguments.token is the access token and tokenSecret is its corresponding secret. profile will contain user profile information provided by the service provider; refer to [User Profile](http://www.passportjs.org/guide/profile/) for additional information.

## Routes

Two routes are required for OAuth authentication. The first route initiates an OAuth transaction and redirects the user to the service provider. The second route is the URL to which the user will be redirected after authenticating with the provider.

// Redirect the user to the OAuth provider for authentication. When

// complete, the provider will redirect the user back to the application at

// /auth/provider/callback

app.get('/auth/provider', passport.authenticate('provider'));

// The OAuth provider has redirected the user back to the application.

// Finish the authentication process by attempting to obtain an access

// token. If authorization was granted, the user will be logged in.

// Otherwise, authentication has failed.

app.get('/auth/provider/callback',

passport.authenticate('provider', { successRedirect: '/',

failureRedirect: '/login' }));

## Link

A link or button can be placed on a web page, which will start the authentication process when clicked.

<a href="/auth/provider">Log In with OAuth Provider</a>

### OAuth 2.0

OAuth 2.0 is the successor to OAuth 1.0, and is designed to overcome perceived shortcomings in the earlier version. The authentication flow is essentially the same. The user is first redirected to the service provider to authorize access. After authorization has been granted, the user is redirected back to the application with a code that can be exchanged for an access token. The application requesting access, known as a client, is identified by an ID and secret.

## Configuration

When using the generic OAuth 2.0 strategy, the client ID, client secret, and endpoints are specified as options.

**var** passport = **require**('passport')

, OAuth2Strategy = **require**('passport-oauth').OAuth2Strategy;

passport.use('provider', **new** OAuth2Strategy({

authorizationURL: 'https://www.provider.com/oauth2/authorize',

tokenURL: 'https://www.provider.com/oauth2/token',

clientID: '123-456-789',

clientSecret: 'shhh-its-a-secret'

callbackURL: 'https://www.example.com/auth/provider/callback'

},

**function**(accessToken, refreshToken, profile, done) {

User.findOrCreate(..., **function**(err, user) {

done(err, user);

});

}

));

The verify callback for OAuth 2.0-based strategies accepts accessToken, refreshToken, and profilearguments. refreshToken can be used to obtain new access tokens, and may be undefined if the provider does not issue refresh tokens. profile will contain user profile information provided by the service provider; refer to [User Profile](http://www.passportjs.org/guide/profile/) for additional information.

## Routes

Two routes are required for OAuth 2.0 authentication. The first route redirects the user to the service provider. The second route is the URL to which the user will be redirected after authenticating with the provider.

// Redirect the user to the OAuth 2.0 provider for authentication. When

// complete, the provider will redirect the user back to the application at

// /auth/provider/callback

app.get('/auth/provider', passport.authenticate('provider'));

// The OAuth 2.0 provider has redirected the user back to the application.

// Finish the authentication process by attempting to obtain an access

// token. If authorization was granted, the user will be logged in.

// Otherwise, authentication has failed.

app.get('/auth/provider/callback',

passport.authenticate('provider', { successRedirect: '/',

failureRedirect: '/login' }));

## Scope

When requesting access using OAuth 2.0, the scope of access is controlled by the scope option.

app.get('/auth/provider',

passport.authenticate('provider', { scope: 'email' })

);

Multiple scopes can be specified as an array.

app.get('/auth/provider',

passport.authenticate('provider', { scope: ['email', 'sms'] })

);

Values for the scope option are provider-specific. Consult the provider's documentation for details regarding supported scopes.

## Link

A link or button can be placed on a web page, which will start the authentication process when clicked.

<a href="/auth/provider">Log In with OAuth 2.0 Provider</a>

# User Profile

When authenticating using a third-party service such as Facebook or Twitter, user profile information will often be available. Each service tends to have a different way of encoding this information. To make integration easier, Passport normalizes profile information to the extent possible.

Normalized profile information conforms to the [contact schema](https://tools.ietf.org/html/draft-smarr-vcarddav-portable-contacts-00) established by [Joseph Smarr][schema-author]. The common fields available are outlined in the following table.

provider {String}

The provider with which the user authenticated (facebook, twitter, etc.).

id {String}

A unique identifier for the user, as generated by the service provider.

displayName {String}

The name of this user, suitable for display.

name {Object}

familyName {String}

The family name of this user, or "last name" in most Western languages.

givenName {String}

The given name of this user, or "first name" in most Western languages.

middleName {String}

The middle name of this user.

emails {Array} [n]

value {String}

The actual email address.

type {String}

The type of email address (home, work, etc.).

photos {Array} [n]

value {String}

The URL of the image.

Note that not all of the above fields are available from every service provider. Some providers may contain additional information not described here. Consult the provider-specific documentation for further details.

# Facebook

The Facebook strategy allows users to log in to a web application using their Facebook account. Internally, Facebook authentication works using OAuth 2.0.

Support for Facebook is implemented by the [passport-facebook](https://github.com/jaredhanson/passport-facebook) module.

## Install

$ npm install passport-facebook

## Configuration

In order to use Facebook authentication, you must first create an app at [Facebook Developers](https://developers.facebook.com/). When created, an app is assigned an App ID and App Secret. Your application must also implement a redirect URL, to which Facebook will redirect users after they have approved access for your application.

**var** passport = **require**('passport')

, FacebookStrategy = **require**('passport-facebook').Strategy;

passport.use(**new** FacebookStrategy({

clientID: FACEBOOK\_APP\_ID,

clientSecret: FACEBOOK\_APP\_SECRET,

callbackURL: "http://www.example.com/auth/facebook/callback"

},

**function**(accessToken, refreshToken, profile, done) {

User.findOrCreate(..., **function**(err, user) {

**if** (err) { **return** done(err); }

done(null, user);

});

}

));

The verify callback for Facebook authentication accepts accessToken, refreshToken, and profilearguments. profile will contain user profile information provided by Facebook; refer to [User Profile](http://www.passportjs.org/guide/profile/) for additional information.

Note: For security reasons, the redirection URL must reside on the same host that is registered with Facebook.

## Routes

Two routes are required for Facebook authentication. The first route redirects the user to Facebook. The second route is the URL to which Facebook will redirect the user after they have logged in.

// Redirect the user to Facebook for authentication. When complete,

// Facebook will redirect the user back to the application at

// /auth/facebook/callback

app.get('/auth/facebook', passport.authenticate('facebook'));

// Facebook will redirect the user to this URL after approval. Finish the

// authentication process by attempting to obtain an access token. If

// access was granted, the user will be logged in. Otherwise,

// authentication has failed.

app.get('/auth/facebook/callback',

passport.authenticate('facebook', { successRedirect: '/',

failureRedirect: '/login' }));

Note that the URL of the callback route matches that of the callbackURL option specified when configuring the strategy.

## Permissions

If your application needs extended permissions, they can be requested by setting the scope option.

app.get('/auth/facebook',

passport.authenticate('facebook', { scope: 'read\_stream' })

);

Multiple permissions can be specified as an array.

app.get('/auth/facebook',

passport.authenticate('facebook', { scope: ['read\_stream', 'publish\_actions'] })

);

## Link

A link or button can be placed on a web page, allowing one-click login with Facebook.

<a href="/auth/facebook">Login with Facebook</a>

# Twitter

The Twitter strategy allows users to sign in to a web application using their Twitter account. Internally, Twitter authentication works using OAuth 1.0a.

Support for Twitter is implemented by the [passport-twitter](https://github.com/jaredhanson/passport-twitter) module.

## Install

$ npm install passport-twitter

## Configuration

In order to use Twitter authentication, you must first create an application at [Twitter Developers](https://dev.twitter.com/). When created, an application is assigned a consumer key and consumer secret. Your application must also implement a callback URL, to which Twitter will redirect users after they have approved access for your application.

**var** passport = **require**('passport')

, TwitterStrategy = **require**('passport-twitter').Strategy;

passport.use(**new** TwitterStrategy({

consumerKey: TWITTER\_CONSUMER\_KEY,

consumerSecret: TWITTER\_CONSUMER\_SECRET,

callbackURL: "http://www.example.com/auth/twitter/callback"

},

**function**(token, tokenSecret, profile, done) {

User.findOrCreate(..., **function**(err, user) {

**if** (err) { **return** done(err); }

done(null, user);

});

}

));

The verify callback for Twitter authentication accepts token, tokenSecret, and profile arguments. profilewill contain user profile information provided by Twitter; refer to [User Profile](http://www.passportjs.org/guide/profile/) for additional information.

## Routes

Two routes are required for Twitter authentication. The first route initiates an OAuth transaction and redirects the user to Twitter. The second route is the URL to which Twitter will redirect the user after they have signed in.

// Redirect the user to Twitter for authentication. When complete, Twitter

// will redirect the user back to the application at

// /auth/twitter/callback

app.get('/auth/twitter', passport.authenticate('twitter'));

// Twitter will redirect the user to this URL after approval. Finish the

// authentication process by attempting to obtain an access token. If

// access was granted, the user will be logged in. Otherwise,

// authentication has failed.

app.get('/auth/twitter/callback',

passport.authenticate('twitter', { successRedirect: '/',

failureRedirect: '/login' }));

Note that the URL of the callback route matches that of the callbackURL option specified when configuring the strategy.

## Link

A link or button can be placed on a web page, allowing one-click sign in with Twitter.

<a href="/auth/twitter">Sign in with Twitter</a>

# Google

The Google strategy allows users to sign in to a web application using their Google account. Google [used to support OpenID internally](https://developers.google.com/identity/protocols/OpenID2Migration#shutdown-timetable), but it now works based on [OpenID Connect](https://developers.google.com/identity/protocols/OpenIDConnect) and supports oAuth 1.0 and oAuth 2.0.

Support for Google is implemented by the [passport-google-oauth](https://github.com/jaredhanson/passport-google-oauth) module.

## Install

$ npm install passport-google-oauth

## Configuration

The Client Id and Client Secret needed to authenticate with Google can be set up from the [Google Developers Console](https://console.developers.google.com/). You may also need to enable Google+ API in the developer console, otherwise user profile data may not be fetched. Google supports authentication with both oAuth 1.0 and oAuth 2.0.

### oAuth 1.0

The Google OAuth 1.0 authentication strategy authenticates users using a Google account and OAuth tokens. The strategy requires a verify callback, which accepts these credentials and calls done providing a user, as well as options specifying a consumer key, consumer secret, and callback URL.

#### Configuration

**var** passport = **require**('passport');

**var** GoogleStrategy = **require**('passport-google-oauth').OAuthStrategy;

// Use the GoogleStrategy within Passport.

// Strategies in passport require a `verify` function, which accept

// credentials (in this case, a token, tokenSecret, and Google profile), and

// invoke a callback with a user object.

passport.use(**new** GoogleStrategy({

consumerKey: GOOGLE\_CONSUMER\_KEY,

consumerSecret: GOOGLE\_CONSUMER\_SECRET,

callbackURL: "http://www.example.com/auth/google/callback"

},

**function**(token, tokenSecret, profile, done) {

User.findOrCreate({ googleId: profile.id }, **function** (err, user) {

**return** done(err, user);

});

}

));

#### Routes

Use passport.authenticate(), specifying the 'google' strategy, to authenticate requests. Authentication with Google requires an extra scope parameter. For information, go [here](https://developers.google.com/identity/protocols/OpenIDConnect#scope-param).

// GET /auth/google

// Use passport.authenticate() as route middleware to authenticate the

// request. The first step in Google authentication will involve redirecting

// the user to google.com. After authorization, Google will redirect the user

// back to this application at /auth/google/callback

app.get('/auth/google',

passport.authenticate('google', { scope: 'https://www.google.com/m8/feeds' });

// GET /auth/google/callback

// Use passport.authenticate() as route middleware to authenticate the

// request. If authentication fails, the user will be redirected back to the

// login page. Otherwise, the primary route function function will be called,

// which, in this example, will redirect the user to the home page.

app.get('/auth/google/callback',

passport.authenticate('google', { failureRedirect: '/login' }),

**function**(req, res) {

res.redirect('/');

});

### oAuth 2.0

The Google OAuth 2.0 authentication strategy authenticates users using a Google account and OAuth 2.0 tokens. The strategy requires a verify callback, which accepts these credentials and calls done providing a user, as well as options specifying a client ID, client secret, and callback URL.

#### Configuration

**var** passport = **require**('passport');

**var** GoogleStrategy = **require**('passport-google-oauth').OAuth2Strategy;

// Use the GoogleStrategy within Passport.

// Strategies in Passport require a `verify` function, which accept

// credentials (in this case, an accessToken, refreshToken, and Google

// profile), and invoke a callback with a user object.

passport.use(**new** GoogleStrategy({

clientID: GOOGLE\_CLIENT\_ID,

clientSecret: GOOGLE\_CLIENT\_SECRET,

callbackURL: "http://www.example.com/auth/google/callback"

},

**function**(accessToken, refreshToken, profile, done) {

User.findOrCreate({ googleId: profile.id }, **function** (err, user) {

**return** done(err, user);

});

}

));

#### Routes

Use passport.authenticate(), specifying the 'google' strategy, to authenticate requests. Authentication with Google requires an extra scope parameter. For information, go [here](https://developers.google.com/identity/protocols/OpenIDConnect#scope-param).

// GET /auth/google

// Use passport.authenticate() as route middleware to authenticate the

// request. The first step in Google authentication will involve

// redirecting the user to google.com. After authorization, Google

// will redirect the user back to this application at /auth/google/callback

app.get('/auth/google',

passport.authenticate('google', { scope: ['https://www.googleapis.com/auth/plus.login'] }));

// GET /auth/google/callback

// Use passport.authenticate() as route middleware to authenticate the

// request. If authentication fails, the user will be redirected back to the

// login page. Otherwise, the primary route function function will be called,

// which, in this example, will redirect the user to the home page.

app.get('/auth/google/callback',

passport.authenticate('google', { failureRedirect: '/login' }),

**function**(req, res) {

res.redirect('/');

});

## Link

A link or button can be placed on a web page, allowing one-click sign in with Google.

<a href="/auth/google">Sign In with Google</a>

# Basic & Digest

Along with defining HTTP's authentication framework, [RFC 2617](http://tools.ietf.org/html/rfc2617) also defined the Basic and Digest authentications schemes. These two schemes both use usernames and passwords as credentials to authenticate users, and are often used to protect API endpoints.

It should be noted that relying on username and password creditials can have adverse security impacts, especially in scenarios where there is not a high degree of trust between the server and client. In these situations, it is recommended to use an authorization framework such as [OAuth 2.0](http://www.passportjs.org/guide/oauth2-api/).

Support for Basic and Digest schemes is provided by the [passport-http](https://github.com/jaredhanson/passport-http) module.

## Install

$ npm install passport-http

### Basic

The Basic scheme uses a username and password to authenticate a user. These credentials are transported in plain text, so it is advised to use HTTPS when implementing this scheme.

## Configuration

passport.use(**new** BasicStrategy(

**function**(username, password, done) {

User.findOne({ username: username }, **function** (err, user) {

**if** (err) { **return** done(err); }

**if** (!user) { **return** done(null, false); }

**if** (!user.validPassword(password)) { **return** done(null, false); }

**return** done(null, user);

});

}

));

The verify callback for Basic authentication accepts username and password arguments.

## Protect Endpoints

app.get('/api/me',

passport.authenticate('basic', { session: false }),

**function**(req, res) {

res.json(req.user);

});

Specify passport.authenticate() with the basic strategy to protect API endpoints. Sessions are not typically needed by APIs, so they can be disabled.

### Digest

The Digest scheme uses a username and password to authenticate a user. Its primary benefit over Basic is that it uses a challenge-response paradigm to avoid sending the password in the clear.

## Configuration

passport.use(**new** DigestStrategy({ qop: 'auth' },

**function**(username, done) {

User.findOne({ username: username }, **function** (err, user) {

**if** (err) { **return** done(err); }

**if** (!user) { **return** done(null, false); }

**return** done(null, user, user.password);

});

},

**function**(params, done) {

// validate nonces as necessary

done(null, true)

}

));

The Digest strategy utilizes two callbacks, the second of which is optional.

The first callback, known as the "secret callback" accepts the username and calls done supplying a user and the corresponding secret password. The password is used to compute a hash, and authentication fails if it does not match that contained in the request.

The second "validate callback" accepts nonce related params, which can be checked to avoid replay attacks.

## Protect Endpoints

app.get('/api/me',

passport.authenticate('digest', { session: false }),

**function**(req, res) {

res.json(req.user);

});

Specify passport.authenticate() with the digest strategy to protect API endpoints. Sessions are not typically needed by APIs, so they can be disabled.

# OAuth

[OAuth](http://oauth.net/) (formally specified by [RFC 5849](http://tools.ietf.org/html/rfc5849)) provides a means for users to grant third-party applications access to their data without exposing their password to those applications.

The protocol greatly improves the security of web applications, in particular, and OAuth has been important in bringing attention to the potential dangers of exposing passwords to external services.

While OAuth 1.0 is still widely used, it has been superseded by [OAuth 2.0](http://www.passportjs.org/guide/oauth2-api/). It is recommended to base new implementations on OAuth 2.0.

When using OAuth to protect API endpoints, there are three distinct steps that that must be performed:

1. The application requests permission from the user for access to protected resources.
2. A token is issued to the application, if permission is granted by the user.
3. The application authenticates using the token to access protected resources.

## Issuing Tokens

[OAuthorize](https://github.com/jaredhanson/oauthorize), a sibling project to Passport, provides a toolkit for implementing OAuth service providers.

The authorization process is a complex sequence that involves authenticating both the requesting application and the user, as well as prompting the user for permission, ensuring that enough detail is provided for the user to make an informed decision.

Additionally, it is up to the implementor to determine what limits can be placed on the application regarding scope of access, as well as subsequently enforcing those limits.

As a toolkit, OAuthorize does not attempt to make implementation decisions. This guide does not cover these issues, but does highly recommend that services deploying OAuth have a complete understanding of the security considerations involved.

## Authenticating Tokens

Once issued, OAuth tokens can be authenticated using the [passport-http-oauth](https://github.com/jaredhanson/passport-http-oauth) module.

## Install

$ npm install passport-http-oauth

## Configuration

passport.use('token', **new** TokenStrategy(

**function**(consumerKey, done) {

Consumer.findOne({ key: consumerKey }, **function** (err, consumer) {

**if** (err) { **return** done(err); }

**if** (!consumer) { **return** done(null, false); }

**return** done(null, consumer, consumer.secret);

});

},

**function**(accessToken, done) {

AccessToken.findOne({ token: accessToken }, **function** (err, token) {

**if** (err) { **return** done(err); }

**if** (!token) { **return** done(null, false); }

Users.findById(token.userId, **function**(err, user) {

**if** (err) { **return** done(err); }

**if** (!user) { **return** done(null, false); }

// fourth argument is optional info. typically used to pass

// details needed to authorize the request (ex: `scope`)

**return** done(null, user, token.secret, { scope: token.scope });

});

});

},

**function**(timestamp, nonce, done) {

// validate the timestamp and nonce as necessary

done(null, true)

}

));

In contrast to other strategies, there are two callbacks required by OAuth. In OAuth, both an identifier for the requesting application and the user-specific token are encoded as credentials.

The first callback is known as the "consumer callback", and is used to find the application making the request, including the secret assigned to it. The second callback is the "token callback", which is used to indentify the user as well as the token's corresponding secret. The secrets supplied by the consumer and token callbacks are used to compute a signature, and authentication fails if it does not match the request signature.

A final "validate callback" is optional, which can be used to prevent replay attacks by checking the timestamp and nonce used in the request.

## Protect Endpoints

app.get('/api/me',

passport.authenticate('token', { session: false }),

**function**(req, res) {

res.json(req.user);

});

Specify passport.authenticate() with the token strategy to protect API endpoints. Sessions are not typically needed by APIs, so they can be disabled.

# OAuth 2.0

OAuth 2.0 (formally specified by [RFC 6749](http://tools.ietf.org/html/rfc6749)) provides an authorization framework which allows users to authorize access to third-party applications. When authorized, the application is issued a token to use as an authentication credential. This has two primary security benefits:

1. The application does not need to store the user's username and password.
2. The token can have a restricted scope (for example: read-only access).

These benefits are particularly important for ensuring the security of web applications, making OAuth 2.0 the predominant standard for API authentication.

When using OAuth 2.0 to protect API endpoints, there are three distinct steps that must be performed:

1. The application requests permission from the user for access to protected resources.
2. A token is issued to the application, if permission is granted by the user.
3. The application authenticates using the token to access protected resources.

## Issuing Tokens

[OAuth2orize](https://github.com/jaredhanson/oauth2orize), a sibling project to Passport, provides a toolkit for implementing OAuth 2.0 authorization servers.

The authorization process is a complex sequence that involves authenticating both the requesting application and the user, as well as prompting the user for permission, ensuring that enough detail is provided for the user to make an informed decision.

Additionally, it is up to the implementor to determine what limits can be placed on the application regarding scope of access, as well as subsequently enforcing those limits.

As a toolkit, OAuth2orize does not attempt to make implementation decisions. This guide does not cover these issues, but does highly recommend that services deploying OAuth 2.0 have a complete understanding of the security considerations involved.

## Authenticating Tokens

OAuth 2.0 provides a framework, in which an arbitrarily extensible set of token types can be issued. In practice, only specific token types have gained widespread use.

## Bearer Tokens

Bearer tokens are the most widely issued type of token in OAuth 2.0. So much so, in fact, that many implementations assume that bearer tokens are the only type of token issued.

Bearer tokens can be authenticated using the [passport-http-bearer](https://github.com/jaredhanson/passport-http-bearer) module.

## Install

$ npm install passport-http-bearer

## Configuration

passport.use(**new** BearerStrategy(

**function**(token, done) {

User.findOne({ token: token }, **function** (err, user) {

**if** (err) { **return** done(err); }

**if** (!user) { **return** done(null, false); }

**return** done(null, user, { scope: 'read' });

});

}

));

The verify callback for bearer tokens accepts the token as an argument. When invoking done, optional info can be passed, which will be set by Passport at req.authInfo. This is typically used to convey the scope of the token, and can be used when making access control checks.

## Protect Endpoints

app.get('/api/me',

passport.authenticate('bearer', { session: false }),

**function**(req, res) {

res.json(req.user);

});

Specify passport.authenticate() with the bearer strategy to protect API endpoints. Sessions are not typically needed by APIs, so they can be disabled.

# API Schemes

The following is a list of strategies that implement authentication schemes used when protecting API endpoints.

| Scheme | Specification | Developer |
| --- | --- | --- |
| [Anonymous](https://github.com/jaredhanson/passport-anonymous) | N/A | [Jared Hanson](https://github.com/jaredhanson) |
| [Bearer](https://github.com/jaredhanson/passport-http-bearer) | [RFC 6750](http://tools.ietf.org/html/rfc6750) | [Jared Hanson](https://github.com/jaredhanson) |
| [Basic](https://github.com/jaredhanson/passport-http) | [RFC 2617](http://tools.ietf.org/html/rfc2617) | [Jared Hanson](https://github.com/jaredhanson) |
| [Digest](https://github.com/jaredhanson/passport-http) | [RFC 2617](http://tools.ietf.org/html/rfc2617) | [Jared Hanson](https://github.com/jaredhanson) |
| [Hash](https://github.com/yuri-karadzhov/passport-hash) | N/A | [Yuri Karadzhov](https://github.com/yuri-karadzhov) |
| [Hawk](https://github.com/jfromaniello/passport-hawk) | [hueniverse/hawk](https://github.com/hueniverse/hawk) | [José F. Romaniello](https://github.com/jfromaniello) |
| [Local API Key](https://github.com/cholalabs/passport-localapikey) | N/A | [Sudhakar Mani](https://github.com/cholalabs) |
| [OAuth](https://github.com/jaredhanson/passport-http-oauth) | [RFC 5849](http://tools.ietf.org/html/rfc5849) | [Jared Hanson](https://github.com/jaredhanson) |
| [OAuth 2.0 Client Password](https://github.com/jaredhanson/passport-oauth2-client-password) | [RFC 6749](http://tools.ietf.org/html/rfc6749#section-2.3.1) | [Jared Hanson](https://github.com/jaredhanson) |
| [OAuth 2.0 JWT Client Assertion](https://github.com/xtuple/passport-oauth2-jwt-bearer) | [draft-jones-oauth-jwt-bearer](http://tools.ietf.org/html/draft-jones-oauth-jwt-bearer) | [xTuple](https://github.com/xtuple) |
| [OAuth 2.0 Public Client](https://github.com/timshadel/passport-oauth2-public-client) | [RFC 6749](http://tools.ietf.org/html/rfc6749) | [Tim Shadel](https://github.com/timshadel) |

# Log In

Passport exposes a login() function on req (also aliased as logIn()) that can be used to establish a login session.

req.login(user, **function**(err) {

**if** (err) { **return** next(err); }

**return** res.redirect('/users/' + req.user.username);

});

When the login operation completes, user will be assigned to req.user.

Note: passport.authenticate() middleware invokes req.login() automatically. This function is primarily used when users sign up, during which req.login() can be invoked to automatically log in the newly registered user.

# Log Out

Passport exposes a logout() function on req (also aliased as logOut()) that can be called from any route handler which needs to terminate a login session. Invoking logout() will remove the req.userproperty and clear the login session (if any).

app.get('/logout', **function**(req, res){

req.logout();

res.redirect('/');

});

# Authorize

An application may need to incorporate information from multiple third-party services. In this case, the application will request the user to "connect", for example, both their Facebook and Twitter accounts.

When this occurs, a user will already be authenticated with the application, and any subsequent third-party accounts merely need to be authorized and associated with the user. Because authentication and authorization in this situation are similar, Passport provides a means to accommodate both.

Authorization is performed by calling passport.authorize(). If authorization is granted, the result provided by the strategy's verify callback will be assigned to req.account. The existing login session and req.userwill be unaffected.

app.get('/connect/twitter',

passport.authorize('twitter-authz', { failureRedirect: '/account' })

);

app.get('/connect/twitter/callback',

passport.authorize('twitter-authz', { failureRedirect: '/account' }),

**function**(req, res) {

**var** user = req.user;

**var** account = req.account;

// Associate the Twitter account with the logged-in user.

account.userId = user.id;

account.save(**function**(err) {

**if** (err) { **return** self.error(err); }

self.redirect('/');

});

}

);

In the callback route, you can see the use of both req.user and req.account. The newly connected account is associated with the logged-in user and saved to the database.

## Configuration

Strategies used for authorization are the same as those used for authentication. However, an application may want to offer both authentication and authorization with the same third-party service. In this case, a named strategy can be used, by overriding the strategy's default name in the call to use().

passport.use('twitter-authz', **new** TwitterStrategy({

consumerKey: TWITTER\_CONSUMER\_KEY,

consumerSecret: TWITTER\_CONSUMER\_SECRET,

callbackURL: "http://www.example.com/connect/twitter/callback"

},

**function**(token, tokenSecret, profile, done) {

Account.findOne({ domain: 'twitter.com', uid: profile.id }, **function**(err, account) {

**if** (err) { **return** done(err); }

**if** (account) { **return** done(null, account); }

**var** account = **new** Account();

account.domain = 'twitter.com';

account.uid = profile.id;

**var** t = { kind: 'oauth', token: token, attributes: { tokenSecret: tokenSecret } };

account.tokens.push(t);

**return** done(null, account);

});

}

));

In the above example, you can see that the twitter-authz strategy is finding or creating an Accountinstance to store Twitter account information. The result will be assigned to req.account, allowing the route handler to associate the account with the authenticated user.

### Association in Verify Callback

One downside to the approach described above is that it requires two instances of the same strategy and supporting routes.

To avoid this, set the strategy's passReqToCallback option to true. With this option enabled, req will be passed as the first argument to the verify callback.

passport.use(**new** TwitterStrategy({

consumerKey: TWITTER\_CONSUMER\_KEY,

consumerSecret: TWITTER\_CONSUMER\_SECRET,

callbackURL: "http://www.example.com/auth/twitter/callback",

passReqToCallback: true

},

**function**(req, token, tokenSecret, profile, done) {

**if** (!req.user) {

// Not logged-in. Authenticate based on Twitter account.

} **else** {

// Logged in. Associate Twitter account with user. Preserve the login

// state by supplying the existing user after association.

// return done(null, req.user);

}

}

));

With req passed as an argument, the verify callback can use the state of the request to tailor the authentication process, handling both authentication and authorization using a single strategy instance and set of routes. For example, if a user is already logged in, the newly "connected" account can be associated. Any additional application-specific properties set on req, including req.session, can be used as well.

1.4) Socket.IO

**Introduction**

Writing a chat application with popular web applications stacks like LAMP (PHP) has traditionally been very hard. It involves polling the server for changes, keeping track of timestamps, and it’s a lot slower than it should be.

Sockets have traditionally been the solution around which most real-time chat systems are architected, providing a bi-directional communication channel between a client and a server.

This means that the server can *push* messages to clients. Whenever you write a chat message, the idea is that the server will get it and push it to all other connected clients.

**The web framework**

The first goal is to setup a simple HTML webpage that serves out a form and a list of messages. We’re going to use the Node.JS web framework express to this end. Make sure [Node.JS](https://nodejs.org/) is installed.

First let’s create a package.json manifest file that describes our project. I recommend you place it in a dedicated empty directory (I’ll call mine chat-example).

|  |
| --- |
| {  "name": "socket-chat-example",  "version": "0.0.1",  "description": "my first socket.io app",  "dependencies": {} } |

Now, in order to easily populate the dependencies with the things we need, we’ll use npm install --save:

|  |
| --- |
| npm install --save express@4.15.2 |

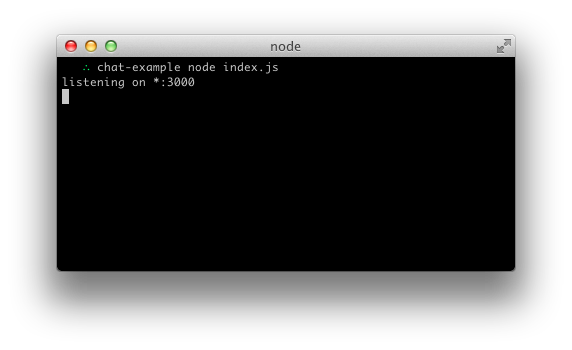
Now that express is installed we can create an index.js file that will setup our application.

|  |
| --- |
| var app = require('express')(); var http = require('http').createServer(app);  app.get('/', function(req, res){  res.send('<h1>Hello world</h1>'); });  http.listen(3000, function(){  console.log('listening on \*:3000'); }); |

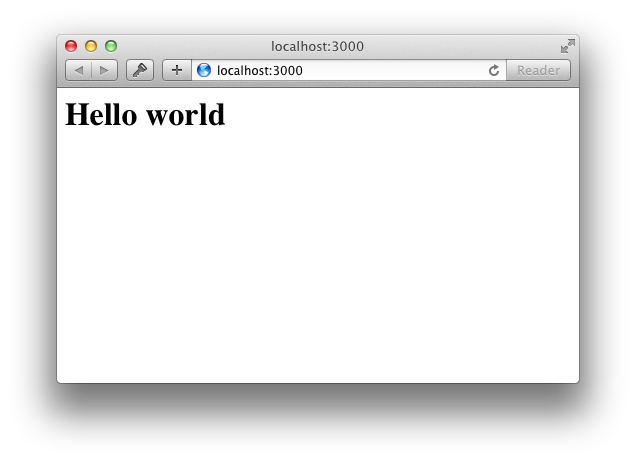
This translates into the following:

* Express initializes app to be a function handler that you can supply to an HTTP server (as seen in line 2).
* We define a route handler / that gets called when we hit our website home.
* We make the http server listen on port 3000.

If you run node index.js you should see the following:



And if you point your browser to http://localhost:3000:



**Serving HTML**

So far in index.js we’re calling res.send and pass it a HTML string. Our code would look very confusing if we just placed our entire application’s HTML there. Instead, we’re going to create a index.html file and serve it.

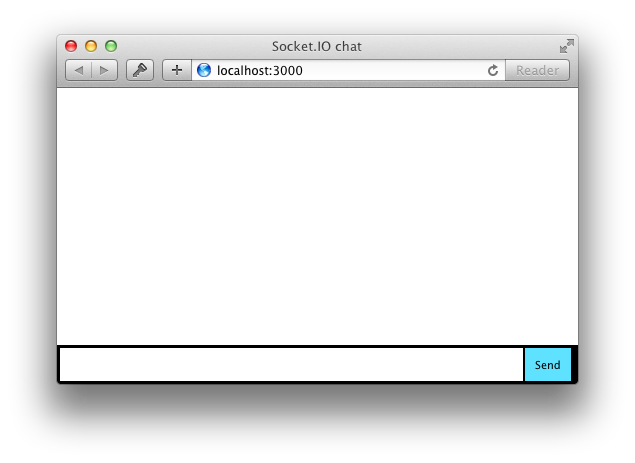
Let’s refactor our route handler to use sendFile instead:

|  |
| --- |
| app.get('/', function(req, res){  res.sendFile(\_\_dirname + '/index.html'); }); |

And populate index.html with the following:

|  |
| --- |
| <!doctype html> <html>  <head>  <title>Socket.IO chat</title>  <style>  \* { margin: 0; padding: 0; box-sizing: border-box; }  body { font: 13px Helvetica, Arial; }  form { background: #000; padding: 3px; position: fixed; bottom: 0; width: 100%; }  form input { border: 0; padding: 10px; width: 90%; margin-right: .5%; }  form button { width: 9%; background: rgb(130, 224, 255); border: none; padding: 10px; }  #messages { list-style-type: none; margin: 0; padding: 0; }  #messages li { padding: 5px 10px; }  #messages li:nth-child(odd) { background: #eee; }  </style>  </head>  <body>  <ul id="messages"></ul>  <form action="">  <input id="m" autocomplete="off" /><button>Send</button>  </form>  </body> </html> |

If you restart the process (by hitting Control+C and running node index again) and refresh the page it should look like this:



**Integrating Socket.IO**

Socket.IO is composed of two parts:

* A server that integrates with (or mounts on) the Node.JS HTTP Server: [socket.io](https://github.com/socketio/socket.io)
* A client library that loads on the browser side: [socket.io-client](https://github.com/socketio/socket.io-client)

During development, socket.io serves the client automatically for us, as we’ll see, so for now we only have to install one module:

|  |
| --- |
| npm install --save socket.io |

That will install the module and add the dependency to package.json. Now let’s edit index.js to add it:

|  |
| --- |
| var app = require('express')(); var http = require('http').createServer(app); var io = require('socket.io')(http);  app.get('/', function(req, res){  res.sendFile(\_\_dirname + '/index.html'); });  io.on('connection', function(socket){  console.log('a user connected'); });  http.listen(3000, function(){  console.log('listening on \*:3000'); }); |

Notice that I initialize a new instance of socket.io by passing the http (the HTTP server) object. Then I listen on the connection event for incoming sockets, and I log it to the console.

Now in index.html I add the following snippet before the </body>:

|  |
| --- |
| <script src="/socket.io/socket.io.js"></script> <script>  var socket = io(); </script> |

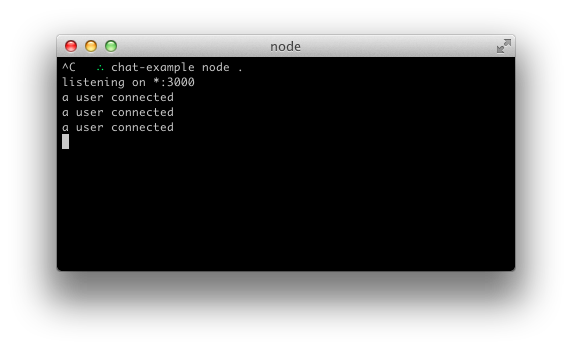
That’s all it takes to load the socket.io-client, which exposes a io global (and the endpoint GET /socket.io/socket.io.js), and then connect.

If you would like to use the local version of the client-side JS file, you can find it at node\_modules/socket.io-client/dist/socket.io.js.

Notice that I’m not specifying any URL when I call io(), since it defaults to trying to connect to the host that serves the page.

If you now reload the server and the website you should see the console print “a user connected”.

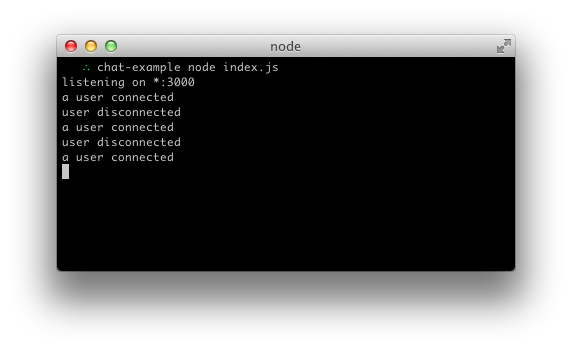
Try opening several tabs, and you’ll see several messages:



Each socket also fires a special disconnect event:

|  |
| --- |
| io.on('connection', function(socket){  console.log('a user connected');  socket.on('disconnect', function(){  console.log('user disconnected');  }); }); |

Then if you refresh a tab several times you can see it in action:



**Emitting events**

The main idea behind Socket.IO is that you can send and receive any events you want, with any data you want. Any objects that can be encoded as JSON will do, and [binary data](https://socket.io/blog/introducing-socket-io-1-0/#binary) is supported too.

Let’s make it so that when the user types in a message, the server gets it as a chat message event. The script section in index.html should now look as follows:

|  |
| --- |
| <script src="/socket.io/socket.io.js"></script> <script src="https://code.jquery.com/jquery-1.11.1.js"></script> <script>  $(function () {  var socket = io();  $('form').submit(function(e){  e.preventDefault(); // prevents page reloading  socket.emit('chat message', $('#m').val());  $('#m').val('');  return false;  });  }); </script> |

And in index.js we print out the chat message event:

|  |
| --- |
| io.on('connection', function(socket){  socket.on('chat message', function(msg){  console.log('message: ' + msg);  }); }); |

The result should be like the following video:

**Broadcasting**

The next goal is for us to emit the event from the server to the rest of the users.

In order to send an event to everyone, Socket.IO gives us the io.emit:

|  |
| --- |
| io.emit('some event', { for: 'everyone' }); |

If you want to send a message to everyone except for a certain socket, we have the broadcast flag:

|  |
| --- |
| io.on('connection', function(socket){  socket.broadcast.emit('hi'); }); |

In this case, for the sake of simplicity we’ll send the message to everyone, including the sender.

|  |
| --- |
| io.on('connection', function(socket){  socket.on('chat message', function(msg){  io.emit('chat message', msg);  }); }); |

And on the client side when we capture a chat message event we’ll include it in the page. The total client-side JavaScript code now amounts to:

|  |
| --- |
| <script>  $(function () {  var socket = io();  $('form').submit(function(e){  e.preventDefault(); // prevents page reloading  socket.emit('chat message', $('#m').val());  $('#m').val('');  return false;  });  socket.on('chat message', function(msg){  $('#messages').append($('<li>').text(msg));  });  }); </script> |

1.5)Validator

A library of string validators and sanitizers.

## Strings only

**This library validates and sanitizes strings only.**

If you're not sure if your input is a string, coerce it using input + ''. Passing anything other than a string is an error.

## Installation and Usage

### Server-side usage

Install the library with npm install validator

#### No ES6

var validator **=** require('validator');

validator.isEmail('foo@bar.com'); *//=> true*

#### ES6

**import** validator **from** 'validator';

Or, import only a subset of the library:

**import** isEmail **from** 'validator/lib/isEmail';

### Client-side usage

The library can be loaded either as a standalone script, or through an [AMD](http://requirejs.org/docs/whyamd.html)-compatible loader

<script **type**="text/javascript" **src**="validator.min.js"></script>

<script **type**="text/javascript">

  validator.isEmail('foo@bar.com'); *//=> true*

</script>

The library can also be installed through [bower](http://bower.io/)

$ bower install validator-js

**Validators**

Here is a list of the validators currently available.

| **Validator** | **Description** |
| --- | --- |
| ***contains(str, seed)*** | check if the string contains the seed. |
| **equals(str, comparison)** | check if the string matches the comparison. |
| **isAfter(str [, date])** | check if the string is a date that's after the specified date (defaults to now). |
| **isAlpha(str [, locale])** | check if the string contains only letters (a-zA-Z).  Locale is one of ['ar', 'ar-AE', 'ar-BH', 'ar-DZ', 'ar-EG', 'ar-IQ', 'ar-JO', 'ar-KW', 'ar-LB', 'ar-LY', 'ar-MA', 'ar-QA', 'ar-QM', 'ar-SA', 'ar-SD', 'ar-SY', 'ar-TN', 'ar-YE', 'bg-BG', 'cs-CZ', 'da-DK', 'de-DE', 'el-GR', 'en-AU', 'en-GB', 'en-HK', 'en-IN', 'en-NZ', 'en-US', 'en-ZA', 'en-ZM', 'es-ES', 'fr-FR', 'hu-HU', 'it-IT', 'ku-IQ', 'nb-NO', 'nl-NL', 'nn-NO', 'pl-PL', 'pt-BR', 'pt-PT', 'ru-RU', 'sl-SI', 'sk-SK', 'sr-RS', 'sr-RS@latin', 'sv-SE', 'tr-TR', 'uk-UA']) and defaults to en-US. Locale list is validator.isAlphaLocales. |
| **isAlphanumeric(str [, locale])** | check if the string contains only letters and numbers.  Locale is one of ['ar', 'ar-AE', 'ar-BH', 'ar-DZ', 'ar-EG', 'ar-IQ', 'ar-JO', 'ar-KW', 'ar-LB', 'ar-LY', 'ar-MA', 'ar-QA', 'ar-QM', 'ar-SA', 'ar-SD', 'ar-SY', 'ar-TN', 'ar-YE', 'bg-BG', 'cs-CZ', 'da-DK', 'de-DE', 'el-GR', 'en-AU', 'en-GB', 'en-HK', 'en-IN', 'en-NZ', 'en-US', 'en-ZA', 'en-ZM', 'es-ES', 'fr-FR', 'hu-HU', 'it-IT', 'ku-IQ', 'nb-NO', 'nl-NL', 'nn-NO', 'pl-PL', 'pt-BR', 'pt-PT', 'ru-RU', 'sl-SI', 'sk-SK', 'sr-RS', 'sr-RS@latin', 'sv-SE', 'tr-TR', 'uk-UA']) and defaults to en-US. Locale list is validator.isAlphanumericLocales. |
| **isAscii(str)** | check if the string contains ASCII chars only. |
| **isBase32(str)** | check if a string is base32 encoded. |
| **isBase64(str)** | check if a string is base64 encoded. |
| **isBefore(str [, date])** | check if the string is a date that's before the specified date. |
| **isBoolean(str)** | check if a string is a boolean. |
| **isByteLength(str [, options])** | check if the string's length (in UTF-8 bytes) falls in a range.  options is an object which defaults to {min:0, max: undefined}. |
| **isCreditCard(str)** | check if the string is a credit card. |
| **isCurrency(str [, options])** | check if the string is a valid currency amount.  options is an object which defaults to {symbol: '$', require\_symbol: false, allow\_space\_after\_symbol: false, symbol\_after\_digits: false, allow\_negatives: true, parens\_for\_negatives: false, negative\_sign\_before\_digits: false, negative\_sign\_after\_digits: false, allow\_negative\_sign\_placeholder: false, thousands\_separator: ',', decimal\_separator: '.', allow\_decimal: true, require\_decimal: false, digits\_after\_decimal: [2], allow\_space\_after\_digits: false}. **Note:** The array digits\_after\_decimal is filled with the exact number of digits allowed not a range, for example a range 1 to 3 will be given as [1, 2, 3]. |
| **isDataURI(str)** | check if the string is a [data uri format](https://developer.mozilla.org/en-US/docs/Web/HTTP/data_URIs). |
| **isMagnetURI(str)** | check if the string is a [magnet uri format](https://en.wikipedia.org/wiki/Magnet_URI_scheme). |
| **isDecimal(str [, options])** | check if the string represents a decimal number, such as 0.1, .3, 1.1, 1.00003, 4.0, etc.  options is an object which defaults to {force\_decimal: false, decimal\_digits: '1,', locale: 'en-US'}  locale determine the decimal separator and is one of ['ar', 'ar-AE', 'ar-BH', 'ar-DZ', 'ar-EG', 'ar-IQ', 'ar-JO', 'ar-KW', 'ar-LB', 'ar-LY', 'ar-MA', 'ar-QA', 'ar-QM', 'ar-SA', 'ar-SD', 'ar-SY', 'ar-TN', 'ar-YE', 'bg-BG', 'cs-CZ', 'da-DK', 'de-DE', 'en-AU', 'en-GB', 'en-HK', 'en-IN', 'en-NZ', 'en-US', 'en-ZA', 'en-ZM', 'es-ES', 'fr-FR', 'hu-HU', 'it-IT', 'ku-IQ', nb-NO', 'nl-NL', 'nn-NO', 'pl-PL', 'pt-BR', 'pt-PT', 'ru-RU', 'sl-SI', 'sr-RS', 'sr-RS@latin', 'sv-SE', 'tr-TR', 'uk-UA']. **Note:** decimal\_digits is given as a range like '1,3', a specific value like '3' or min like '1,'. |
| **isDivisibleBy(str, number)** | check if the string is a number that's divisible by another. |
| **isEmail(str [, options])** | check if the string is an email.  options is an object which defaults to { allow\_display\_name: false, require\_display\_name: false, allow\_utf8\_local\_part: true, require\_tld: true, allow\_ip\_domain: false, domain\_specific\_validation: false }. If allow\_display\_name is set to true, the validator will also match Display Name <email-address>. If require\_display\_name is set to true, the validator will reject strings without the format Display Name <email-address>. If allow\_utf8\_local\_part is set to false, the validator will not allow any non-English UTF8 character in email address' local part. If require\_tld is set to false, e-mail addresses without having TLD in their domain will also be matched. If ignore\_max\_length is set to true, the validator will not check for the standard max length of an email. If allow\_ip\_domain is set to true, the validator will allow IP addresses in the host part. If domain\_specific\_validation is true, some additional validation will be enabled, e.g. disallowing certain syntactically valid email addresses that are rejected by GMail. |
| **isEmpty(str [, options])** | check if the string has a length of zero.  options is an object which defaults to { ignore\_whitespace:false }. |
| **isFQDN(str [, options])** | check if the string is a fully qualified domain name (e.g. domain.com).  options is an object which defaults to { require\_tld: true, allow\_underscores: false, allow\_trailing\_dot: false }. |
| **isFloat(str [, options])** | check if the string is a float.  options is an object which can contain the keys min, max, gt, and/or lt to validate the float is within boundaries (e.g. { min: 7.22, max: 9.55 }) it also has locale as an option.  min and max are equivalent to 'greater or equal' and 'less or equal', respectively while gt and lt are their strict counterparts.  locale determine the decimal separator and is one of ['ar', 'ar-AE', 'ar-BH', 'ar-DZ', 'ar-EG', 'ar-IQ', 'ar-JO', 'ar-KW', 'ar-LB', 'ar-LY', 'ar-MA', 'ar-QA', 'ar-QM', 'ar-SA', 'ar-SD', 'ar-SY', 'ar-TN', 'ar-YE', 'bg-BG', 'cs-CZ', 'da-DK', 'de-DE', 'en-AU', 'en-GB', 'en-HK', 'en-IN', 'en-NZ', 'en-US', 'en-ZA', 'en-ZM', 'es-ES', 'fr-FR', 'hu-HU', 'it-IT', 'nb-NO', 'nl-NL', 'nn-NO', 'pl-PL', 'pt-BR', 'pt-PT', 'ru-RU', 'sl-SI', 'sr-RS', 'sr-RS@latin', 'sv-SE', 'tr-TR', 'uk-UA']. Locale list is validator.isFloatLocales. |
| **isFullWidth(str)** | check if the string contains any full-width chars. |
| **isHalfWidth(str)** | check if the string contains any half-width chars. |
| **isHash(str, algorithm)** | check if the string is a hash of type algorithm.  Algorithm is one of ['md4', 'md5', 'sha1', 'sha256', 'sha384', 'sha512', 'ripemd128', 'ripemd160', 'tiger128', 'tiger160', 'tiger192', 'crc32', 'crc32b'] |
| **isHexColor(str)** | check if the string is a hexadecimal color. |
| **isHexadecimal(str)** | check if the string is a hexadecimal number. |
| **isIdentityCard(str [, locale])** | check if the string is a valid identity card code.  locale is one of ['ES'] OR 'any'. If 'any' is used, function will check if any of the locals match.  Defaults to 'any'. |
| **isIP(str [, version])** | check if the string is an IP (version 4 or 6). |
| **isIPRange(str)** | check if the string is an IP Range(version 4 only). |
| **isISBN(str [, version])** | check if the string is an ISBN (version 10 or 13). |
| **isISSN(str [, options])** | check if the string is an [ISSN](https://en.wikipedia.org/wiki/International_Standard_Serial_Number).  options is an object which defaults to { case\_sensitive: false, require\_hyphen: false }. If case\_sensitiveis true, ISSNs with a lowercase 'x' as the check digit are rejected. |
| **isISIN(str)** | check if the string is an [ISIN](https://en.wikipedia.org/wiki/International_Securities_Identification_Number) (stock/security identifier). |
| **isISO8601(str)** | check if the string is a valid [ISO 8601](https://en.wikipedia.org/wiki/ISO_8601) date; for additional checks for valid dates, e.g. invalidates dates like 2009-02-29, pass options object as a second parameter with options.strict = true. |
| **isRFC3339(str)** | check if the string is a valid [RFC 3339](https://tools.ietf.org/html/rfc3339) date. |
| **isISO31661Alpha2(str)** | check if the string is a valid [ISO 3166-1 alpha-2](https://en.wikipedia.org/wiki/ISO_3166-1_alpha-2) officially assigned country code. |
| **isISO31661Alpha3(str)** | check if the string is a valid [ISO 3166-1 alpha-3](https://en.wikipedia.org/wiki/ISO_3166-1_alpha-3) officially assigned country code. |
| **isISRC(str)** | check if the string is a [ISRC](https://en.wikipedia.org/wiki/International_Standard_Recording_Code). |
| **isIn(str, values)** | check if the string is in a array of allowed values. |
| **isInt(str [, options])** | check if the string is an integer.  options is an object which can contain the keys min and/or max to check the integer is within boundaries (e.g. { min: 10, max: 99 }). options can also contain the key allow\_leading\_zeroes, which when set to false will disallow integer values with leading zeroes (e.g. { allow\_leading\_zeroes: false }). Finally, options can contain the keys gt and/or lt which will enforce integers being greater than or less than, respectively, the value provided (e.g. {gt: 1, lt: 4} for a number between 1 and 4). |
| **isJSON(str)** | check if the string is valid JSON (note: uses JSON.parse). |
| **isJWT(str)** | check if the string is valid JWT token. |
| **isLatLong(str)** | check if the string is a valid latitude-longitude coordinate in the format lat,long or lat, long. |
| **isLength(str [, options])** | check if the string's length falls in a range.  options is an object which defaults to {min:0, max: undefined}. Note: this function takes into account surrogate pairs. |
| **isLowercase(str)** | check if the string is lowercase. |
| **isMACAddress(str)** | check if the string is a MAC address.  options is an object which defaults to {no\_colons: false}. If no\_colons is true, the validator will allow MAC addresses without the colons. |
| **isMD5(str)** | check if the string is a MD5 hash. |
| **isMimeType(str)** | check if the string matches to a valid [MIME type](https://en.wikipedia.org/wiki/Media_type) format |
| **isMobilePhone(str [, locale [, options]])** | check if the string is a mobile phone number,  (locale is either an array of locales (e.g ['sk-SK', 'sr-RS']) OR one of ['ar-AE', 'ar-DZ', 'ar-EG', 'ar-IQ', ar-JO', 'ar-KW', 'ar-SA', 'ar-SY', 'ar-TN', 'be-BY', 'bg-BG', 'bn-BD', 'cs-CZ', 'de-DE', 'da-DK', 'el-GR', 'en-AU', 'en-CA', 'en-GB', 'en-GH', 'en-HK', 'en-IE', 'en-IN', 'en-KE', 'en-MU', en-NG', 'en-NZ', 'en-RW', 'en-SG', 'en-UG', 'en-US', 'en-TZ', 'en-ZA', 'en-ZM', 'en-PK', 'es-ES', 'es-MX', 'es-PY', 'es-UY', 'et-EE', 'fa-IR', 'fi-FI', 'fr-FR', 'he-IL', 'hu-HU', 'id-ID', 'it-IT', 'ja-JP', 'kk-KZ', 'ko-KR', 'lt-LT', 'ms-MY', 'nb-NO', 'nn-NO', 'pl-PL', 'pt-PT', 'pt-BR', 'ro-RO', 'ru-RU', 'sl-SI', 'sk-SK', 'sr-RS', 'sv-SE', 'th-TH', 'tr-TR', 'uk-UA', 'vi-VN', 'zh-CN', 'zh-HK', 'zh-TW'] OR defaults to 'any'. If 'any' or a falsey value is used, function will check if any of the locales match).  options is an optional object that can be supplied with the following keys: strictMode, if this is set to true, the mobile phone number must be supplied with the country code and therefore must start with +. Locale list is validator.isMobilePhoneLocales. |
| **isMongoId(str)** | check if the string is a valid hex-encoded representation of a [MongoDB ObjectId](http://docs.mongodb.org/manual/reference/object-id/). |
| **isMultibyte(str)** | check if the string contains one or more multibyte chars. |
| **isNumeric(str [, options])** | check if the string contains only numbers.  options is an object which defaults to {no\_symbols: false}. If no\_symbols is true, the validator will reject numeric strings that feature a symbol (e.g. +, -, or .). |
| **isPort(str)** | check if the string is a valid port number. |
| **isPostalCode(str, locale)** | check if the string is a postal code,  (locale is one of [ 'AD', 'AT', 'AU', 'BE', 'BG', 'CA', 'CH', 'CZ', 'DE', 'DK', 'DZ', 'EE', 'ES', 'FI', 'FR', 'GB', 'GR', 'HR', 'HU', 'ID', 'IL', 'IN', 'IS', 'IT', 'JP', 'KE', 'LI', 'LT', 'LU', 'LV', 'MX', 'NL', 'NO', 'PL', 'PT', 'RO', 'RU', 'SA', 'SE', 'SI', 'TN', 'TW', 'UA', 'US', 'ZA', 'ZM' ] OR 'any'. If 'any' is used, function will check if any of the locals match. Locale list is validator.isPostalCodeLocales.). |
| **isSurrogatePair(str)** | check if the string contains any surrogate pairs chars. |
| **isURL(str [, options])** | check if the string is an URL.  options is an object which defaults to { protocols: ['http','https','ftp'], require\_tld: true, require\_protocol: false, require\_host: true, require\_valid\_protocol: true, allow\_underscores: false, host\_whitelist: false, host\_blacklist: false, allow\_trailing\_dot: false, allow\_protocol\_relative\_urls: false, disallow\_auth: false }. |
| **isUUID(str [, version])** | check if the string is a UUID (version 3, 4 or 5). |
| **isUppercase(str)** | check if the string is uppercase. |
| **isVariableWidth(str)** | check if the string contains a mixture of full and half-width chars. |
| **isWhitelisted(str, chars)** | checks characters if they appear in the whitelist. |
| **matches(str, pattern [, modifiers])** | check if string matches the pattern.  Either matches('foo', /foo/i) or matches('foo', 'foo', 'i'). |

**Sanitizers**

Here is a list of the sanitizers currently available.

| **Sanitizer** | **Description** |
| --- | --- |
| **blacklist(input, chars)** | remove characters that appear in the blacklist. The characters are used in a RegExp and so you will need to escape some chars, e.g. blacklist(input, '\\[\\]'). |
| **escape(input)** | replace <, >, &, ', " and / with HTML entities. |
| **unescape(input)** | replaces HTML encoded entities with <, >, &, ', " and /. |
| **ltrim(input [, chars])** | trim characters from the left-side of the input. |
| **normalizeEmail(email [, options])** | canonicalizes an email address. (This doesn't validate that the input is an email, if you want to validate the email use isEmail beforehand)  options is an object with the following keys and default values:   * *all\_lowercase: true* - Transforms the local part (before the @ symbol) of all email addresses to lowercase. Please note that this may violate RFC 5321, which gives providers the possibility to treat the local part of email addresses in a case sensitive way (although in practice most - yet not all - providers don't). The domain part of the email address is always lowercased, as it's case insensitive per RFC 1035. * *gmail\_lowercase: true* - GMail addresses are known to be case-insensitive, so this switch allows lowercasing them even when *all\_lowercase* is set to false. Please note that when *all\_lowercase* is true, GMail addresses are lowercased regardless of the value of this setting. * *gmail\_remove\_dots: true*: Removes dots from the local part of the email address, as GMail ignores them (e.g. "john.doe" and "johndoe" are considered equal). * *gmail\_remove\_subaddress: true*: Normalizes addresses by removing "sub-addresses", which is the part following a "+" sign (e.g. ["foo+bar@gmail.com](mailto:%22foo+bar@gmail.com)" becomes ["foo@gmail.com](mailto:%22foo@gmail.com)"). * *gmail\_convert\_googlemaildotcom: true*: Converts addresses with domain @googlemail.com to @gmail.com, as they're equivalent. * *outlookdotcom\_lowercase: true* - Outlook.com addresses (including Windows Live and Hotmail) are known to be case-insensitive, so this switch allows lowercasing them even when *all\_lowercase* is set to false. Please note that when *all\_lowercase* is true, Outlook.com addresses are lowercased regardless of the value of this setting. * *outlookdotcom\_remove\_subaddress: true*: Normalizes addresses by removing "sub-addresses", which is the part following a "+" sign (e.g. ["foo+bar@outlook.com](mailto:%22foo+bar@outlook.com)" becomes ["foo@outlook.com](mailto:%22foo@outlook.com)"). * *yahoo\_lowercase: true* - Yahoo Mail addresses are known to be case-insensitive, so this switch allows lowercasing them even when *all\_lowercase* is set to false. Please note that when *all\_lowercase* is true, Yahoo Mail addresses are lowercased regardless of the value of this setting. * *yahoo\_remove\_subaddress: true*: Normalizes addresses by removing "sub-addresses", which is the part following a "-" sign (e.g. ["foo-bar@yahoo.com](mailto:%22foo-bar@yahoo.com)" becomes ["foo@yahoo.com](mailto:%22foo@yahoo.com)"). * *icloud\_lowercase: true* - iCloud addresses (including MobileMe) are known to be case-insensitive, so this switch allows lowercasing them even when *all\_lowercase* is set to false. Please note that when *all\_lowercase* is true, iCloud addresses are lowercased regardless of the value of this setting. * *icloud\_remove\_subaddress: true*: Normalizes addresses by removing "sub-addresses", which is the part following a "+" sign (e.g. ["foo+bar@icloud.com](mailto:%22foo+bar@icloud.com)" becomes ["foo@icloud.com](mailto:%22foo@icloud.com)"). |
| **rtrim(input [, chars])** | trim characters from the right-side of the input. |
| **stripLow(input [, keep\_new\_lines])** | remove characters with a numerical value < 32 and 127, mostly control characters. If keep\_new\_lines is true, newline characters are preserved (\n and \r, hex 0xA and 0xD). Unicode-safe in JavaScript. |
| **toBoolean(input [, strict])** | convert the input string to a boolean. Everything except for '0', 'false' and '' returns true. In strict mode only '1' and 'true' return true. |
| **toDate(input)** | convert the input string to a date, or null if the input is not a date. |
| **toFloat(input)** | convert the input string to a float, or NaN if the input is not a float. |
| **toInt(input [, radix])** | convert the input string to an integer, or NaN if the input is not an integer. |
| **trim(input [, chars])** | trim characters (whitespace by default) from both sides of the input. |
| **whitelist(input, chars)** | remove characters that do not appear in the whitelist. The characters are used in a RegExp and so you will need to escape some chars, e.g. whitelist(input, '\\[\\]'). |

1.6) Multer:

Multer is a node.js middleware for handling multipart/form-data, which is primarily used for uploading files. It is written on top of [busboy](https://github.com/mscdex/busboy) for maximum efficiency.

**NOTE**: Multer will not process any form which is not multipart (multipart/form-data).

## Translations

This README is also available in other languages:

* [简体中文](https://github.com/expressjs/multer/blob/master/doc/README-zh-cn.md) (Chinese)
* [한국어](https://github.com/expressjs/multer/blob/master/doc/README-ko.md) (Korean)

## Installation

$ npm install --save multer

## Usage

Multer adds a body object and a file or files object to the request object. The bodyobject contains the values of the text fields of the form, the file or files object contains the files uploaded via the form.

Basic usage example:

Don't forget the enctype="multipart/form-data" in your form.

<form **action**="/profile" **method**="post" **enctype**="multipart/form-data">

  <input **type**="file" **name**="avatar" />

</form>

var express **=** require('express')

var multer  **=** require('multer')

var upload **=** multer({ dest**:** 'uploads/' })

var app **=** express()

app.post('/profile', upload.single('avatar'), function (req, res, next) {

*// req.file is the `avatar` file*

*// req.body will hold the text fields, if there were any*

})

app.post('/photos/upload', upload.array('photos', 12), function (req, res, next) {

*// req.files is array of `photos` files*

*// req.body will contain the text fields, if there were any*

})

var cpUpload **=** upload.fields([{ name**:** 'avatar', maxCount**:** 1 }, { name**:** 'gallery', maxCount**:** 8 }])

app.post('/cool-profile', cpUpload, function (req, res, next) {

*// req.files is an object (String -> Array) where fieldname is the key, and the value is array of files*

*//*

*// e.g.*

*//  req.files['avatar'][0] -> File*

*//  req.files['gallery'] -> Array*

*//*

*// req.body will contain the text fields, if there were any*

})

In case you need to handle a text-only multipart form, you should use the .none() method:

var express **=** require('express')

var app **=** express()

var multer  **=** require('multer')

var upload **=** multer()

app.post('/profile', upload.none(), function (req, res, next) {

*// req.body contains the text fields*

})

## API

### File information

Each file contains the following information:

| **Key** | **Description** | **Note** |
| --- | --- | --- |
| fieldname | Field name specified in the form |  |
| originalname | Name of the file on the user's computer |  |
| encoding | Encoding type of the file |  |
| mimetype | Mime type of the file |  |
| size | Size of the file in bytes |  |
| destination | The folder to which the file has been saved | DiskStorage |
| filename | The name of the file within the destination | DiskStorage |
| path | The full path to the uploaded file | DiskStorage |
| buffer | A Buffer of the entire file | MemoryStorage |

### multer(opts)

Multer accepts an options object, the most basic of which is the dest property, which tells Multer where to upload the files. In case you omit the options object, the files will be kept in memory and never written to disk.

By default, Multer will rename the files so as to avoid naming conflicts. The renaming function can be customized according to your needs.

The following are the options that can be passed to Multer.

| **Key** | **Description** |
| --- | --- |
| dest or storage | Where to store the files |
| fileFilter | Function to control which files are accepted |
| limits | Limits of the uploaded data |
| preservePath | Keep the full path of files instead of just the base name |

In an average web app, only dest might be required, and configured as shown in the following example.

var upload **=** multer({ dest**:** 'uploads/' })

If you want more control over your uploads, you'll want to use the storage option instead of dest. Multer ships with storage engines DiskStorage and MemoryStorage; More engines are available from third parties.

#### .single(fieldname)

Accept a single file with the name fieldname. The single file will be stored in req.file.

#### .array(fieldname[, maxCount])

Accept an array of files, all with the name fieldname. Optionally error out if more than maxCount files are uploaded. The array of files will be stored in req.files.

#### .fields(fields)

Accept a mix of files, specified by fields. An object with arrays of files will be stored in req.files.

fields should be an array of objects with name and optionally a maxCount. Example:

[

  { name**:** 'avatar', maxCount**:** 1 },

  { name**:** 'gallery', maxCount**:** 8 }

]

#### .none()

Accept only text fields. If any file upload is made, error with code "LIMIT\_UNEXPECTED\_FILE" will be issued.

#### .any()

Accepts all files that comes over the wire. An array of files will be stored in req.files.

**WARNING:** Make sure that you always handle the files that a user uploads. Never add multer as a global middleware since a malicious user could upload files to a route that you didn't anticipate. Only use this function on routes where you are handling the uploaded files.

### storage

#### DiskStorage

The disk storage engine gives you full control on storing files to disk.

var storage **=** multer.diskStorage({

  destination**:** function (req, file, cb) {

    cb(null, '/tmp/my-uploads')

  },

  filename**:** function (req, file, cb) {

    cb(null, file.fieldname **+** '-' **+** Date.now())

  }

})

var upload **=** multer({ storage**:** storage })

There are two options available, destination and filename. They are both functions that determine where the file should be stored.

destination is used to determine within which folder the uploaded files should be stored. This can also be given as a string (e.g. '/tmp/uploads'). If no destination is given, the operating system's default directory for temporary files is used.

**Note:** You are responsible for creating the directory when providing destination as a function. When passing a string, multer will make sure that the directory is created for you.

filename is used to determine what the file should be named inside the folder. If no filename is given, each file will be given a random name that doesn't include any file extension.

**Note:** Multer will not append any file extension for you, your function should return a filename complete with an file extension.

Each function gets passed both the request (req) and some information about the file (file) to aid with the decision.

Note that req.body might not have been fully populated yet. It depends on the order that the client transmits fields and files to the server.

#### MemoryStorage

The memory storage engine stores the files in memory as Buffer objects. It doesn't have any options.

var storage **=** multer.memoryStorage()

var upload **=** multer({ storage**:** storage })

When using memory storage, the file info will contain a field called buffer that contains the entire file.

**WARNING**: Uploading very large files, or relatively small files in large numbers very quickly, can cause your application to run out of memory when memory storage is used.

### limits

An object specifying the size limits of the following optional properties. Multer passes this object into busboy directly, and the details of the properties can be found on [busboy's page](https://github.com/mscdex/busboy#busboy-methods).

The following integer values are available:

| **Key** | **Description** | **Default** |
| --- | --- | --- |
| fieldNameSize | Max field name size | 100 bytes |
| fieldSize | Max field value size | 1MB |
| fields | Max number of non-file fields | Infinity |
| fileSize | For multipart forms, the max file size (in bytes) | Infinity |
| files | For multipart forms, the max number of file fields | Infinity |
| parts | For multipart forms, the max number of parts (fields + files) | Infinity |
| headerPairs | For multipart forms, the max number of header key=>value pairs to parse | 2000 |

Specifying the limits can help protect your site against denial of service (DoS) attacks.

### fileFilter

Set this to a function to control which files should be uploaded and which should be skipped. The function should look like this:

function fileFilter (req, file, cb) {

*// The function should call `cb` with a boolean*

*// to indicate if the file should be accepted*

*// To reject this file pass `false`, like so:*

  cb(null, false)

*// To accept the file pass `true`, like so:*

  cb(null, true)

*// You can always pass an error if something goes wrong:*

  cb(**new** Error('I don\'t have a clue!'))

}

## Error handling

When encountering an error, Multer will delegate the error to Express. You can display a nice error page using [the standard express way](http://expressjs.com/guide/error-handling.html).

If you want to catch errors specifically from Multer, you can call the middleware function by yourself. Also, if you want to catch only [the Multer errors](https://github.com/expressjs/multer/blob/master/lib/make-error.js#L1-L9), you can use the MulterError class that is attached to the multer object itself (e.g. err instanceof multer.MulterError).

var multer **=** require('multer')

var upload **=** multer().single('avatar')

app.post('/profile', function (req, res) {

  upload(req, res, function (err) {

**if** (err **instanceof** multer.MulterError) {

*// A Multer error occurred when uploading.*

    } **else** **if** (err) {

*// An unknown error occurred when uploading.*

    }

*// Everything went fine.*

  })

})

1.7) Stripe:

## Usage

The package needs to be configured with your account's secret key which is available in your [Stripe Dashboard](https://dashboard.stripe.com/account/apikeys). Require it with the key's value:

const stripe **=** require('stripe')('sk\_test\_...');

const customer **=** **await** stripe.customers.create({

  email**:** 'customer@example.com',

});

Or using ES modules, this looks more like:

**import** Stripe **from** 'stripe';

const stripe **=** Stripe('sk\_test\_...');

*//…*

On older versions of Node, you can use [promises](https://www.npmjs.com/package/stripe#using-promises) or [callbacks](https://www.npmjs.com/package/stripe#using-callbacks) instead of async/await.

### Usage with TypeScript

Stripe does not currently maintain typings for this package, but there are community typings available from DefinitelyTyped.

To install:

npm install --dev @types/stripe

To use:

*// Note `\* as` and `new Stripe` for TypeScript:*

**import** \* **as** Stripe **from** 'stripe';

const stripe **=** **new** Stripe('sk\_test\_...');

const customer**:** Promise<

  Stripe.customers.ICustomer

**>** **=** stripe.customers.create(*/\* ... \*/*);

### Using Promises

Every method returns a chainable promise which can be used instead of a regular callback:

*// Create a new customer and then a new charge for that customer:*

stripe.customers

  .create({

    email**:** 'foo-customer@example.com',

  })

  .then((customer) => {

**return** stripe.customers.createSource(customer.id, {

      source**:** 'tok\_visa',

    });

  })

  .then((source) => {

**return** stripe.charges.create({

      amount**:** 1600,

      currency**:** 'usd',

      customer**:** source.customer,

    });

  })

  .then((charge) => {

*// New charge created on a new customer*

  })

  .catch((err) => {

*// Deal with an error*

  });

### Using callbacks

On versions of Node.js prior to v7.9:

var stripe **=** require('stripe')('sk\_test\_...');

stripe.customers.create(

  {

    email**:** 'customer@example.com',

  },

  function(err, customer) {

**if** (err) {

*// Deal with an error (will be `null` if no error occurred).*

    }

*// Do something with created customer object*

    console.log(customer.id);

  }

);

### Configuring Timeout

Request timeout is configurable (the default is Node's default of 120 seconds):

stripe.setTimeout(20000); *// in ms (this is 20 seconds)*

### Configuring For Connect

A per-request Stripe-Account header for use with [Stripe Connect](https://stripe.com/connect) can be added to any method:

*// Retrieve the balance for a connected account:*

stripe.balance

  .retrieve({

    stripe\_account**:** 'acct\_foo',

  })

  .then((balance) => {

*// The balance object for the connected account*

  })

  .catch((err) => {

*// Error*

  });

### Configuring a Proxy

An [https-proxy-agent](https://github.com/TooTallNate/node-https-proxy-agent) can be configured with setHttpAgent.

To use stripe behind a proxy you can pass to sdk:

**if** (process.env.http\_proxy) {

  const ProxyAgent **=** require('https-proxy-agent');

  stripe.setHttpAgent(**new** ProxyAgent(process.env.http\_proxy));

}

### Network retries

Automatic network retries can be enabled with setMaxNetworkRetries. This will retry requests n times with exponential backoff if they fail due to an intermittent network problem.[Idempotency keys](https://stripe.com/docs/api/idempotent_requests) are added where appropriate to prevent duplication.

*// Retry a request once before giving up*

stripe.setMaxNetworkRetries(1);

### Examining Responses

Some information about the response which generated a resource is available with the lastResponse property:

charge.lastResponse.requestId; *// see: https://stripe.com/docs/api/node#request\_ids*

charge.lastResponse.statusCode;

### request and response events

The Stripe object emits request and response events. You can use them like this:

const stripe **=** require('stripe')('sk\_test\_...');

const onRequest **=** (request) => {

*// Do something.*

};

*// Add the event handler function:*

stripe.on('request', onRequest);

*// Remove the event handler function:*

stripe.off('request', onRequest);

#### request object

{

  api\_version**:** 'latest',

  account**:** 'acct\_TEST',       *// Only present if provided*

  idempotency\_key**:** 'abc123',  *// Only present if provided*

  method**:** 'POST',

  path**:** '/v1/charges'

}

#### response object

{

  api\_version**:** 'latest',

  account**:** 'acct\_TEST',       *// Only present if provided*

  idempotency\_key**:** 'abc123',  *// Only present if provided*

  method**:** 'POST',

  path**:** '/v1/charges',

  status**:** 402,

  request\_id**:** 'req\_Ghc9r26ts73DRf',

  elapsed**:** 445                *// Elapsed time in milliseconds*

}

### Webhook signing

Stripe can optionally sign the webhook events it sends to your endpoint, allowing you to validate that they were not sent by a third-party. You can read more about it [here](https://stripe.com/docs/webhooks#signatures).

Please note that you must pass the raw request body, exactly as received from Stripe, to the constructEvent() function; this will not work with a parsed (i.e., JSON) request body.

You can find an example of how to use this with [Express](https://expressjs.com/) in the [examples/webhook-signing](https://github.com/stripe/stripe-node/blob/HEAD/examples/webhook-signing)folder, but here's what it looks like:

const event **=** stripe.webhooks.constructEvent(

  webhookRawBody,

  webhookStripeSignatureHeader,

  webhookSecret

);

#### Testing Webhook signing

You can use stripe.webhooks.generateTestHeaderString to mock webhook events that come from Stripe:

const payload **=** {

  id**:** 'evt\_test\_webhook',

  object**:** 'event',

};

const payloadString **=** JSON.stringify(payload, null, 2);

const secret **=** 'whsec\_test\_secret';

const header **=** stripe.webhooks.generateTestHeaderString({

  payload**:** payloadString,

  secret,

});

const event **=** stripe.webhooks.constructEvent(payloadString, header, secret);

*// Do something with mocked signed event*

expect(event.id).to.equal(payload.id);

### Writing a Plugin

If you're writing a plugin that uses the library, we'd appreciate it if you identified using stripe.setAppInfo():

stripe.setAppInfo({

  name**:** 'MyAwesomePlugin',

  version**:** '1.2.34', *// Optional*

  url**:** 'https://myawesomeplugin.info', *// Optional*

});

This information is passed along when the library makes calls to the Stripe API.

### Auto-pagination

As of stripe-node 6.11.0, you may auto-paginate list methods. We provide a few different APIs for this to aid with a variety of node versions and styles.

#### Async iterators (for-await-of)

If you are in a Node environment that has support for [async iteration](https://github.com/tc39/proposal-async-iteration#the-async-iteration-statement-for-await-of), such as Node 10+ or [babel](https://babeljs.io/docs/en/babel-plugin-transform-async-generator-functions), the following will auto-paginate:

**for** **await** (const customer **of** stripe.customers.list()) {

  doSomething(customer);

**if** (shouldStop()) {

**break**;

  }

}

#### autoPagingEach

If you are in a Node environment that has support for await, such as Node 7.9 and greater, you may pass an async function to .autoPagingEach:

**await** stripe.customers.list().autoPagingEach(async (customer) => {

**await** doSomething(customer);

**if** (shouldBreak()) {

**return** false;

  }

});

console.log('Done iterating.');

Equivalently, without await, you may return a Promise, which can resolve to false to break:

stripe.customers

  .list()

  .autoPagingEach((customer) => {

**return** doSomething(customer).then(() => {

**if** (shouldBreak()) {

**return** false;

      }

    });

  })

  .then(() => {

    console.log('Done iterating.');

  })

  .catch(handleError);

If you prefer callbacks to promises, you may also use a next callback and a second onDonecallback:

stripe.customers.list().autoPagingEach(

  function onItem(customer, next) {

    doSomething(customer, function(err, result) {

**if** (shouldStop(result)) {

        next(false); *// Passing `false` breaks out of the loop.*

      } **else** {

        next();

      }

    });

  },

  function onDone(err) {

**if** (err) {

      console.error(err);

    } **else** {

      console.log('Done iterating.');

    }

  }

);

If your onItem function does not accept a next callback parameter or return a Promise, the return value is used to decide whether to continue (false breaks, anything else continues).

#### autoPagingToArray

This is a convenience for cases where you expect the number of items to be relatively small; accordingly, you must pass a limit option to prevent runaway list growth from consuming too much memory.

Returns a promise of an array of all items across pages for a list request.

const allNewCustomers **=** **await** stripe.customers

  .list({created**:** {gt**:** lastMonth}})

  .autoPagingToArray({limit**:** 10000});

2) Front-End Packages:

2.1) Axios

## Installing

Using npm:

$ npm install axios

Using bower:

$ bower install axios

Using yarn:

$ yarn add axios

Using cdn:

<script src="https://unpkg.com/axios/dist/axios.min.js"></script>

## Example

Performing a GET request

const axios = require('axios');

// Make a request for a user with a given ID

axios.get('/user?ID=12345')

.then(function (response) {

// handle success

console.log(response);

})

.catch(function (error) {

// handle error

console.log(error);

})

.finally(function () {

// always executed

});

// Optionally the request above could also be done as

axios.get('/user', {

params: {

ID: 12345

}

})

.then(function (response) {

console.log(response);

})

.catch(function (error) {

console.log(error);

})

.then(function () {

// always executed

});

// Want to use async/await? Add the `async` keyword to your outer function/method.

async function getUser() {

try {

const response = await axios.get('/user?ID=12345');

console.log(response);

} catch (error) {

console.error(error);

}

}

**NOTE:** async/await is part of ECMAScript 2017 and is not supported in Internet Explorer and older browsers, so use with caution.

Performing a POST request

axios.post('/user', {

firstName: 'Fred',

lastName: 'Flintstone'

})

.then(function (response) {

console.log(response);

})

.catch(function (error) {

console.log(error);

});

Performing multiple concurrent requests

function getUserAccount() {

return axios.get('/user/12345');

}

function getUserPermissions() {

return axios.get('/user/12345/permissions');

}

axios.all([getUserAccount(), getUserPermissions()])

.then(axios.spread(function (acct, perms) {

// Both requests are now complete

}));

## axios API

Requests can be made by passing the relevant config to axios.

##### axios(config)

// Send a POST request

axios({

method: 'post',

url: '/user/12345',

data: {

firstName: 'Fred',

lastName: 'Flintstone'

}

});

// GET request for remote image

axios({

method: 'get',

url: 'http://bit.ly/2mTM3nY',

responseType: 'stream'

})

.then(function (response) {

response.data.pipe(fs.createWriteStream('ada\_lovelace.jpg'))

});

##### axios(url[, config])

// Send a GET request (default method)

axios('/user/12345');

### Request method aliases

For convenience aliases have been provided for all supported request methods.

##### axios.request(config)

##### axios.get(url[, config])

##### axios.delete(url[, config])

##### axios.head(url[, config])

##### axios.options(url[, config])

##### axios.post(url[, data[, config]])

##### axios.put(url[, data[, config]])

##### axios.patch(url[, data[, config]])

###### NOTE

When using the alias methods url, method, and data properties don't need to be specified in config.

### Concurrency

Helper functions for dealing with concurrent requests.

##### axios.all(iterable)

##### axios.spread(callback)

### Creating an instance

You can create a new instance of axios with a custom config.

##### axios.create([config])

const instance = axios.create({

baseURL: 'https://some-domain.com/api/',

timeout: 1000,

headers: {'X-Custom-Header': 'foobar'}

});

### Instance methods

The available instance methods are listed below. The specified config will be merged with the instance config.

##### axios#request(config)

##### axios#get(url[, config])

##### axios#delete(url[, config])

##### axios#head(url[, config])

##### axios#options(url[, config])

##### axios#post(url[, data[, config]])

##### axios#put(url[, data[, config]])

##### axios#patch(url[, data[, config]])

##### axios#getUri([config])

## Request Config

These are the available config options for making requests. Only the url is required. Requests will default to GET if methodis not specified.

{

// `url` is the server URL that will be used for the request

url: '/user',

// `method` is the request method to be used when making the request

method: 'get', // default

// `baseURL` will be prepended to `url` unless `url` is absolute.

// It can be convenient to set `baseURL` for an instance of axios to pass relative URLs

// to methods of that instance.

baseURL: 'https://some-domain.com/api/',

// `transformRequest` allows changes to the request data before it is sent to the server

// This is only applicable for request methods 'PUT', 'POST', 'PATCH' and 'DELETE'

// The last function in the array must return a string or an instance of Buffer, ArrayBuffer,

// FormData or Stream

// You may modify the headers object.

transformRequest: [function (data, headers) {

// Do whatever you want to transform the data

return data;

}],

// `transformResponse` allows changes to the response data to be made before

// it is passed to then/catch

transformResponse: [function (data) {

// Do whatever you want to transform the data

return data;

}],

// `headers` are custom headers to be sent

headers: {'X-Requested-With': 'XMLHttpRequest'},

// `params` are the URL parameters to be sent with the request

// Must be a plain object or a URLSearchParams object

params: {

ID: 12345

},

// `paramsSerializer` is an optional function in charge of serializing `params`

// (e.g. https://www.npmjs.com/package/qs, http://api.jquery.com/jquery.param/)

paramsSerializer: function (params) {

return Qs.stringify(params, {arrayFormat: 'brackets'})

},

// `data` is the data to be sent as the request body

// Only applicable for request methods 'PUT', 'POST', and 'PATCH'

// When no `transformRequest` is set, must be of one of the following types:

// - string, plain object, ArrayBuffer, ArrayBufferView, URLSearchParams

// - Browser only: FormData, File, Blob

// - Node only: Stream, Buffer

data: {

firstName: 'Fred'

},

// `timeout` specifies the number of milliseconds before the request times out.

// If the request takes longer than `timeout`, the request will be aborted.

timeout: 1000, // default is `0` (no timeout)

// `withCredentials` indicates whether or not cross-site Access-Control requests

// should be made using credentials

withCredentials: false, // default

// `adapter` allows custom handling of requests which makes testing easier.

// Return a promise and supply a valid response (see lib/adapters/README.md).

adapter: function (config) {

/\* ... \*/

},

// `auth` indicates that HTTP Basic auth should be used, and supplies credentials.

// This will set an `Authorization` header, overwriting any existing

// `Authorization` custom headers you have set using `headers`.

// Please note that only HTTP Basic auth is configurable through this parameter.

// For Bearer tokens and such, use `Authorization` custom headers instead.

auth: {

username: 'janedoe',

password: 's00pers3cret'

},

// `responseType` indicates the type of data that the server will respond with

// options are: 'arraybuffer', 'document', 'json', 'text', 'stream'

// browser only: 'blob'

responseType: 'json', // default

// `responseEncoding` indicates encoding to use for decoding responses

// Note: Ignored for `responseType` of 'stream' or client-side requests

responseEncoding: 'utf8', // default

// `xsrfCookieName` is the name of the cookie to use as a value for xsrf token

xsrfCookieName: 'XSRF-TOKEN', // default

// `xsrfHeaderName` is the name of the http header that carries the xsrf token value

xsrfHeaderName: 'X-XSRF-TOKEN', // default

// `onUploadProgress` allows handling of progress events for uploads

onUploadProgress: function (progressEvent) {

// Do whatever you want with the native progress event

},

// `onDownloadProgress` allows handling of progress events for downloads

onDownloadProgress: function (progressEvent) {

// Do whatever you want with the native progress event

},

// `maxContentLength` defines the max size of the http response content in bytes allowed

maxContentLength: 2000,

// `validateStatus` defines whether to resolve or reject the promise for a given

// HTTP response status code. If `validateStatus` returns `true` (or is set to `null`

// or `undefined`), the promise will be resolved; otherwise, the promise will be

// rejected.

validateStatus: function (status) {

return status >= 200 && status < 300; // default

},

// `maxRedirects` defines the maximum number of redirects to follow in node.js.

// If set to 0, no redirects will be followed.

maxRedirects: 5, // default

// `socketPath` defines a UNIX Socket to be used in node.js.

// e.g. '/var/run/docker.sock' to send requests to the docker daemon.

// Only either `socketPath` or `proxy` can be specified.

// If both are specified, `socketPath` is used.

socketPath: null, // default

// `httpAgent` and `httpsAgent` define a custom agent to be used when performing http

// and https requests, respectively, in node.js. This allows options to be added like

// `keepAlive` that are not enabled by default.

httpAgent: new http.Agent({ keepAlive: true }),

httpsAgent: new https.Agent({ keepAlive: true }),

// 'proxy' defines the hostname and port of the proxy server.

// You can also define your proxy using the conventional `http\_proxy` and

// `https\_proxy` environment variables. If you are using environment variables

// for your proxy configuration, you can also define a `no\_proxy` environment

// variable as a comma-separated list of domains that should not be proxied.

// Use `false` to disable proxies, ignoring environment variables.

// `auth` indicates that HTTP Basic auth should be used to connect to the proxy, and

// supplies credentials.

// This will set an `Proxy-Authorization` header, overwriting any existing

// `Proxy-Authorization` custom headers you have set using `headers`.

proxy: {

host: '127.0.0.1',

port: 9000,

auth: {

username: 'mikeymike',

password: 'rapunz3l'

}

},

// `cancelToken` specifies a cancel token that can be used to cancel the request

// (see Cancellation section below for details)

cancelToken: new CancelToken(function (cancel) {

})

}

## Response Schema

The response for a request contains the following information.

{

// `data` is the response that was provided by the server

data: {},

// `status` is the HTTP status code from the server response

status: 200,

// `statusText` is the HTTP status message from the server response

statusText: 'OK',

// `headers` the headers that the server responded with

// All header names are lower cased

headers: {},

// `config` is the config that was provided to `axios` for the request

config: {},

// `request` is the request that generated this response

// It is the last ClientRequest instance in node.js (in redirects)

// and an XMLHttpRequest instance the browser

request: {}

}

When using then, you will receive the response as follows:

axios.get('/user/12345')

.then(function (response) {

console.log(response.data);

console.log(response.status);

console.log(response.statusText);

console.log(response.headers);

console.log(response.config);

});

When using catch, or passing a [rejection callback](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then) as second parameter of then, the response will be available through the error object as explained in the [Handling Errors](https://github.com/axios/axios#handling-errors) section.

## Config Defaults

You can specify config defaults that will be applied to every request.

### Global axios defaults

axios.defaults.baseURL = 'https://api.example.com';

axios.defaults.headers.common['Authorization'] = AUTH\_TOKEN;

axios.defaults.headers.post['Content-Type'] = 'application/x-www-form-urlencoded';

### Custom instance defaults

// Set config defaults when creating the instance

const instance = axios.create({

baseURL: 'https://api.example.com'

});

// Alter defaults after instance has been created

instance.defaults.headers.common['Authorization'] = AUTH\_TOKEN;

### Config order of precedence

Config will be merged with an order of precedence. The order is library defaults found in [lib/defaults.js](https://github.com/axios/axios/blob/master/lib/defaults.js#L28), then defaultsproperty of the instance, and finally config argument for the request. The latter will take precedence over the former. Here's an example.

// Create an instance using the config defaults provided by the library

// At this point the timeout config value is `0` as is the default for the library

const instance = axios.create();

// Override timeout default for the library

// Now all requests using this instance will wait 2.5 seconds before timing out

instance.defaults.timeout = 2500;

// Override timeout for this request as it's known to take a long time

instance.get('/longRequest', {

timeout: 5000

});

## Interceptors

You can intercept requests or responses before they are handled by then or catch.

// Add a request interceptor

axios.interceptors.request.use(function (config) {

// Do something before request is sent

return config;

}, function (error) {

// Do something with request error

return Promise.reject(error);

});

// Add a response interceptor

axios.interceptors.response.use(function (response) {

// Do something with response data

return response;

}, function (error) {

// Do something with response error

return Promise.reject(error);

});

If you may need to remove an interceptor later you can.

const myInterceptor = axios.interceptors.request.use(function () {/\*...\*/});

axios.interceptors.request.eject(myInterceptor);

You can add interceptors to a custom instance of axios.

const instance = axios.create();

instance.interceptors.request.use(function () {/\*...\*/});

## Handling Errors

axios.get('/user/12345')

.catch(function (error) {

if (error.response) {

// The request was made and the server responded with a status code

// that falls out of the range of 2xx

console.log(error.response.data);

console.log(error.response.status);

console.log(error.response.headers);

} else if (error.request) {

// The request was made but no response was received

// `error.request` is an instance of XMLHttpRequest in the browser and an instance of

// http.ClientRequest in node.js

console.log(error.request);

} else {

// Something happened in setting up the request that triggered an Error

console.log('Error', error.message);

}

console.log(error.config);

});

You can define a custom HTTP status code error range using the validateStatus config option.

axios.get('/user/12345', {

validateStatus: function (status) {

return status < 500; // Reject only if the status code is greater than or equal to 500

}

})

## Cancellation

You can cancel a request using a cancel token.

The axios cancel token API is based on the withdrawn [cancelable promises proposal](https://github.com/tc39/proposal-cancelable-promises).

You can create a cancel token using the CancelToken.source factory as shown below:

const CancelToken = axios.CancelToken;

const source = CancelToken.source();

axios.get('/user/12345', {

cancelToken: source.token

}).catch(function (thrown) {

if (axios.isCancel(thrown)) {

console.log('Request canceled', thrown.message);

} else {

// handle error

}

});

axios.post('/user/12345', {

name: 'new name'

}, {

cancelToken: source.token

})

// cancel the request (the message parameter is optional)

source.cancel('Operation canceled by the user.');

You can also create a cancel token by passing an executor function to the CancelToken constructor:

const CancelToken = axios.CancelToken;

let cancel;

axios.get('/user/12345', {

cancelToken: new CancelToken(function executor(c) {

// An executor function receives a cancel function as a parameter

cancel = c;

})

});

// cancel the request

cancel();

Note: you can cancel several requests with the same cancel token.

## Using application/x-www-form-urlencoded format

By default, axios serializes JavaScript objects to JSON. To send data in the application/x-www-form-urlencoded format instead, you can use one of the following options.

### Browser

In a browser, you can use the [URLSearchParams](https://developer.mozilla.org/en-US/docs/Web/API/URLSearchParams) API as follows:

const params = new URLSearchParams();

params.append('param1', 'value1');

params.append('param2', 'value2');

axios.post('/foo', params);

Note that URLSearchParams is not supported by all browsers (see [caniuse.com](http://www.caniuse.com/#feat=urlsearchparams)), but there is a [polyfill](https://github.com/WebReflection/url-search-params) available (make sure to polyfill the global environment).

Alternatively, you can encode data using the [qs](https://github.com/ljharb/qs) library:

const qs = require('qs');

axios.post('/foo', qs.stringify({ 'bar': 123 }));

Or in another way (ES6),

import qs from 'qs';

const data = { 'bar': 123 };

const options = {

method: 'POST',

headers: { 'content-type': 'application/x-www-form-urlencoded' },

data: qs.stringify(data),

url,

};

axios(options);

### Node.js

In node.js, you can use the [querystring](https://nodejs.org/api/querystring.html) module as follows:

const querystring = require('querystring');

axios.post('http://something.com/', querystring.stringify({ foo: 'bar' }));

You can also use the [qs](https://github.com/ljharb/qs) library.

###### NOTE

The qs library is preferable if you need to stringify nested objects, as the querystring method has known issues with that use case (<https://github.com/nodejs/node-v0.x-archive/issues/1665>).

## Semver

Until axios reaches a 1.0 release, breaking changes will be released with a new minor version. For example 0.5.1, and 0.5.4 will have the same API, but 0.6.0 will have breaking changes.

## Promises

axios depends on a native ES6 Promise implementation to be [supported](http://caniuse.com/promises). If your environment doesn't support ES6 Promises, you can [polyfill](https://github.com/jakearchibald/es6-promise).

## TypeScript

axios includes [TypeScript](http://typescriptlang.org/) definitions.

import axios from 'axios';

axios.get('/user?ID=12345');

2.2)jwt-decode

## Usage

var token **=** 'eyJ0eXAiO.../// jwt token';

var decoded **=** jwt\_decode(token);

console.log(decoded);

*/\* prints:*

*\* { foo: "bar",*

*\*   exp: 1393286893,*

*\*   iat: 1393268893  }*

*\*/*

**Note:** A falsy token will throw an error.

Can also be used with [browserify](http://browserify.org/) or [webpack](http://webpack.github.io/) by doing npm install jwt-decode and requiring:

var jwtDecode **=** require('jwt-decode');

## Polymer Web Component

Can also be installed and used with [Polymer-based wrapper](https://github.com/firmfirm/f-jwt-decode).

2.3) Native Base

**1. What is NativeBase?**

[NativeBase](https://nativebase.io/) is a sleek, ingenious and dynamic front-end framework created by passionate React Loving team at [Geekyants.com](https://geekyants.com/) to build cross platform Android & iOS mobile apps using ready to use generic components of React Native.

**2. Why NativeBase?**

What is really great with [NativeBase](https://nativebase.io/) is that you can use shared UI cross-platform components, which will drastically increase your productivity. When using NativeBase, you can use any native third-party libraries out of the box.

**Recommended by** *Microsoft*

If you're looking to get started with React Native + CodePush, and are looking for an awesome starter kit, you should check out Native Starter Pro - [Microsoft's react-native-code-push repo](https://github.com/Microsoft/react-native-code-push#example-apps--starters)

**Recommended by**[Awesome React Native](https://github.com/jondot/awesome-react-native)

NativeBase added into the list of [Frameworks](https://github.com/jondot/awesome-react-native#frameworks) of Awesome React Native and are also used by many other React lovers across the world.

**4. Getting Started**

**a. Setup with pure React Native app**

*Install NativeBase*

npm install native**-**base **--**save

*Install Peer Dependencies*  
The peer dependencies included from any npm packages does not automatically get installed. Your application will not depend on it explicitly.

react**-**native link

You've successfully setup [NativeBase](https://nativebase.io/) with your [React Native](https://facebook.github.io/react-native/) app. Your React Native app is now all set to run on iOS and Android simulator.

**b. Setup with Expo**

Expo helps you make React Native apps with no build configuration. It works on macOS, Windows, and Linux.   
Refer this link for additional information on [Expo](https://docs.expo.io/)

*Install NativeBase*

npm install native**-**base **--**save

**Note**   
[NativeBase](https://nativebase.io/) uses some custom fonts that can be loaded using **Font.loadAsync**. Check out the [Expo Font documentation](https://docs.expo.io/versions/latest/sdk/font/).   
Syntax

*// At the top of your file*

**import** { Font } **from** 'expo';

**import** { Ionicons } **from** '@expo/vector-icons';

*// Later on in your component*

async componentDidMount() {

**await** Font.loadAsync({

    'Roboto'**:** require('native-base/Fonts/Roboto.ttf'),

    'Roboto\_medium'**:** require('native-base/Fonts/Roboto\_medium.ttf'),

**...**Ionicons.font,

  });

}

Check out the [KitchenSink](https://expo.io/@geekyants/nativebasekitchensink) with Expo for an example of the implementation.  
Find the [KitchenSink repo here](https://github.com/GeekyAnts/NativeBase-KitchenSink/tree/CRNA)

**c. Setup with ignite-native-base-boilerplate**

You can run the following command to create the boilerplate, provided you have [Ignite CLI](https://github.com/infinitered/ignite)installed.

ignite new appname --boilerplate native-base-boilerplate

Go to app location

cd appname

For iOS run

react-native run-ios

For Android run

react-native run-android

Refer [ignite-native-base-boilerplate](https://github.com/GeekyAnts/ignite-native-base-boilerplate) page for additional information

**5. Components**

[NativeBase](https://nativebase.io/) is made from effective building blocks referred to as components. The Components are constructed in pure [React Native](https://github.com/facebook/react-native) platform along with some JavaScript functionality with rich set of customisable properties. These components allow you to quickly build the perfect interface.

**6. NativeBase for Web**

NativeBase is now available for React web lovers. Check the [demo](https://nativebase.io/kitchen-sink-web-app/) Find the repo [here](https://github.com/GeekyAnts/NativeBase-KitchenSink/tree/web-support)

**7. Compatibility Versions**

| **NativeBase** | **React Native** |
| --- | --- |
| v0.1.1 | v0.22 to v0.23 |
| v0.2.0 to v0.3.1 | v0.24 to v0.25 |
| v0.4.6 to v0.4.9 | v0.26.0 - v0.27.1 |
| v0.5.0 to v0.5.15 | v0.26.0 - v0.37.0 |
| v0.5.16 to v0.5.20 | v0.38.0 - v0.39.0 |
| v2.0.0-alpha1 to v2.1.3 | v0.38.0 to v0.43.0 |
| v2.1.4 to v2.1.5 | v0.44.0 to v0.45.0 |
| v2.2.0 | v0.44.0 to v0.45.0 |
| v2.2.1 | v0.46.0 and above |
| v2.3.0 to 2.6.1 | v0.46.0 and above *(does not support React 16.0.0-alpha.13)* |
| v2.7.0 | v0.56.0 and above |

**8.**[**React Native Seed**](https://reactnativeseed.com/)

React Native Seed provides you React Native starter kits for your base app with the technologies that you love.

Based on the feedback we received from our users, people had trouble sorting out the right boilerplate for them with the desired technologies and contacted us to enquire. We realized that many people were particular about the technologies they want in the app and that a minimal, neat solution was required to solve this, and hence, React Native Seed.

React Native Seed is for learners and professionals alike, those who want to experiment, learn all aspects and those who already know enough, just want a starter kit to quickly start working on their project.

**9. NativeBase Market**

Having tried with the free version, [Native Starter Kit](https://github.com/start-react/native-starter-kit) and appreciate our product?

Get on the mobile fast track with the featured apps of [NativeBase](https://nativebase.io/), to build high-quality iOS and Android mobile apps.  
A [marketplace](https://market.nativebase.io/) for premium React Native app themes to build high-quality iOS and Android mobile apps.

2.4) Redux

# Getting Started with Redux

Redux is a predictable state container for JavaScript apps.

It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test. On top of that, it provides a great developer experience, such as [live code editing combined with a time traveling debugger](https://github.com/reduxjs/redux-devtools).

You can use Redux together with [React](https://reactjs.org/), or with any other view library. It is tiny (2kB, including dependencies), but has a large ecosystem of addons available.

## Installation

Redux is available as a package on NPM for use with a module bundler or in a Node application:

npm install --save redux

**Copy**

It is also available as a precompiled UMD package that defines a window.Redux global variable. The UMD package can be used as a [<script> tag](https://unpkg.com/redux/dist/redux.js) directly.

For more details, see the [Installation](https://redux.js.org/introduction/installation) page.

## Redux Starter Kit

Redux itself is small and unopinionated. We also have a separate package called [**redux-starter-kit**](https://redux-starter-kit.js.org/), which includes some opinionated defaults that help you use Redux more effectively.

It helps simplify a lot of common use cases, including [store setup](https://redux-starter-kit.js.org/api/configureStore), [creating reducers and writing immutable update logic](https://redux-starter-kit.js.org/api/createreducer), and even [creating entire "slices" of state at once](https://redux-starter-kit.js.org/api/createslice).

Whether you're a brand new Redux user setting up your first project, or an experienced user who wants to simplify an existing application, [**redux-starter-kit**](https://redux-starter-kit.js.org/) can help you make your Redux code better.

## Basic Example

The whole state of your app is stored in an object tree inside a single store.  
The only way to change the state tree is to emit an action, an object describing what happened.  
To specify how the actions transform the state tree, you write pure reducers.

That's it!

**import** { createStore } **from** 'redux'

/\*\*

\* This is a reducer, a pure function with (state, action) => state signature.

\* It describes how an action transforms the state into the next state.

\*

\* The shape of the state is up to you: it can be a primitive, an array, an object,

\* or even an Immutable.js data structure. The only important part is that you should

\* not mutate the state object, but return a new object if the state changes.

\*

\* In this example, we use a `switch` statement and strings, but you can use a helper that

\* follows a different convention (such as function maps) if it makes sense for your

\* project.

\*/

**function** **counter**(state = 0, action) {

**switch** (action.type) {

**case** 'INCREMENT':

**return** state + 1

**case** 'DECREMENT':

**return** state - 1

**default**:

**return** state

}

}

// Create a Redux store holding the state of your app.

// Its API is { subscribe, dispatch, getState }.

**let** store = createStore(counter)

// You can use subscribe() to update the UI in response to state changes.

// Normally you'd use a view binding library (e.g. React Redux) rather than subscribe() directly.

// However it can also be handy to persist the current state in the localStorage.

store.subscribe(() => console.log(store.getState()))

// The only way to mutate the internal state is to dispatch an action.

// The actions can be serialized, logged or stored and later replayed.

store.dispatch({ type: 'INCREMENT' })

// 1

store.dispatch({ type: 'INCREMENT' })

// 2

store.dispatch({ type: 'DECREMENT' })

// 1

**Copy**

Instead of mutating the state directly, you specify the mutations you want to happen with plain objects called actions. Then you write a special function called a reducer to decide how every action transforms the entire application's state.

In a typical Redux app, there is just a single store with a single root reducing function. As your app grows, you split the root reducer into smaller reducers independently operating on the different parts of the state tree. This is exactly like how there is just one root component in a React app, but it is composed out of many small components.

This architecture might seem like an overkill for a counter app, but the beauty of this pattern is how well it scales to large and complex apps. It also enables very powerful developer tools, because it is possible to trace every mutation to the action that caused it. You can record user sessions and reproduce them just by replaying every action.

## Examples

The Redux repository contains several example projects demonstrating various aspects of how to use Redux. Almost all examples have a corresponding CodeSandbox sandbox. This is an interactive version of the code that you can play with online.

* [**Counter Vanilla**](https://redux.js.org/introduction/examples#counter-vanilla): [Source](https://github.com/reduxjs/redux/tree/master/examples/counter-vanilla)
* [**Counter**](https://redux.js.org/introduction/examples#counter): [Source](https://github.com/reduxjs/redux/tree/master/examples/counter) | [Sandbox](https://codesandbox.io/s/github/reduxjs/redux/tree/master/examples/counter)
* [**Todos**](https://redux.js.org/introduction/examples#todos): [Source](https://github.com/reduxjs/redux/tree/master/examples/todos) | [Sandbox](https://codesandbox.io/s/github/reduxjs/redux/tree/master/examples/todos)
* [**Todos with Undo**](https://redux.js.org/introduction/examples#todos-with-undo): [Source](https://github.com/reduxjs/redux/tree/master/examples/todos-with-undo) | [Sandbox](https://codesandbox.io/s/github/reduxjs/redux/tree/master/examples/todos-with-undo)
* [**TodoMVC**](https://redux.js.org/introduction/examples#todomvc): [Source](https://github.com/reduxjs/redux/tree/master/examples/todomvc) | [Sandbox](https://codesandbox.io/s/github/reduxjs/redux/tree/master/examples/todomvc)
* [**Shopping Cart**](https://redux.js.org/introduction/examples#shopping-cart): [Source](https://github.com/reduxjs/redux/tree/master/examples/shopping-cart) | [Sandbox](https://codesandbox.io/s/github/reduxjs/redux/tree/master/examples/shopping-cart)
* [**Tree View**](https://redux.js.org/introduction/examples#tree-view): [Source](https://github.com/reduxjs/redux/tree/master/examples/tree-view) | [Sandbox](https://codesandbox.io/s/github/reduxjs/redux/tree/master/examples/tree-view)
* [**Async**](https://redux.js.org/introduction/examples#async): [Source](https://github.com/reduxjs/redux/tree/master/examples/async) | [Sandbox](https://codesandbox.io/s/github/reduxjs/redux/tree/master/examples/async)
* [**Universal**](https://redux.js.org/introduction/examples#universal): [Source](https://github.com/reduxjs/redux/tree/master/examples/universal)
* [**Real World**](https://redux.js.org/introduction/examples#real-world): [Source](https://github.com/reduxjs/redux/tree/master/examples/real-world) | [Sandbox](https://codesandbox.io/s/github/reduxjs/redux/tree/master/examples/real-world)

## Learn Redux

We have a variety of resources available to help you learn Redux, no matter what your background or learning style is.

### Just the Basics

If you're brand new to Redux and want to understand the basic concepts, see:

* The [**Motivation**](https://redux.js.org/introduction/motivation) behind building Redux, the [**Core Concepts**](https://redux.js.org/introduction/core-concepts), and the [**Three Principles**](https://redux.js.org/introduction/three-principles).
* The [**basic tutorial in the Redux docs**](https://redux.js.org/basics/basic-tutorial)
* Redux creator Dan Abramov's **free**[**"Getting Started with Redux" video series**](https://egghead.io/series/getting-started-with-redux) on Egghead.io
* Redux co-maintainer Mark Erikson's [**"Redux Fundamentals" slideshow**](http://blog.isquaredsoftware.com/2018/03/presentation-reactathon-redux-fundamentals/) and [**list of suggested resources for learning Redux**](http://blog.isquaredsoftware.com/2017/12/blogged-answers-learn-redux/)
* If you learn best by looking at code and playing with it, check out our list of [**Redux example applications**](https://redux.js.org/introduction/examples), available as separate projects in the Redux repo, and also as interactive online examples on CodeSandbox.
* The [**Redux Tutorials**](https://github.com/markerikson/react-redux-links/blob/master/redux-tutorials.md) section of the [**React/Redux links list**](https://github.com/markerikson/react-redux-links). Here's a top list of our recommended tutorials:
  + Dave Ceddia's posts [What Does Redux Do? (and when should you use it?)](https://daveceddia.com/what-does-redux-do/) and [How Redux Works: A Counter-Example](https://daveceddia.com/how-does-redux-work/) are a great intro to the basics of Redux and how to use it with React, as is this post on [React and Redux: An Introduction](http://jakesidsmith.com/blog/post/2017-11-18-redux-and-react-an-introduction/).
  + Valentino Gagliardi's post [React Redux Tutorial for Beginners: Learning Redux in 2018](https://www.valentinog.com/blog/react-redux-tutorial-beginners/) is an excellent extended introduction to many aspects of using Redux.
  + The CSS Tricks article [Leveling Up with React: Redux](https://css-tricks.com/learning-react-redux/) covers the Redux basics well.
  + This [DevGuides: Introduction to Redux](http://devguides.io/redux/) tutorial covers several aspects of Redux, including actions, reducers, usage with React, and middleware.

### Intermediate Concepts

Once you've picked up the basics of working with actions, reducers, and the store, you may have questions about topics like working with asynchronous logic and AJAX requests, connecting a UI framework like React to your Redux store, and setting up an application to use Redux:

* The [**"Advanced" docs section**](https://redux.js.org/advanced/advanced-tutorial) covers working with async logic, middleware, routing.
* The Redux docs [**"Learning Resources"**](https://redux.js.org/introduction/learning-resources) page points to recommended articles on a variety of Redux-related topics.
* Sophie DeBenedetto's 8-part [**Building a Simple CRUD App with React + Redux**](http://www.thegreatcodeadventure.com/building-a-simple-crud-app-with-react-redux-part-1/) series shows how to put together a basic CRUD app from scratch.

### Real-World Usage

Going from a TodoMVC app to a real production application can be a big jump, but we've got plenty of resources to help:

* Redux creator Dan Abramov's [**free "Building React Applications with Idiomatic Redux" video series**](https://egghead.io/courses/building-react-applications-with-idiomatic-redux) builds on his first video series and covers topics like middleware, routing, and persistence.
* The [**Redux FAQ**](https://redux.js.org/faq) answers many common questions about how to use Redux, and the [**"Recipes" docs section**](https://redux.js.org/recipes/recipe-index) has information on handling derived data, testing, structuring reducer logic, and reducing boilerplate.
* Redux co-maintainer Mark Erikson's [**"Practical Redux" tutorial series**](http://blog.isquaredsoftware.com/series/practical-redux/) demonstrates real-world intermediate and advanced techniques for working with React and Redux (also available as [**an interactive course on Educative.io**](https://www.educative.io/collection/5687753853370368/5707702298738688)).
* The [**React/Redux links list**](https://github.com/markerikson/react-redux-links) has categorized articles on working with [reducers and selectors](https://github.com/markerikson/react-redux-links/blob/master/redux-reducers-selectors.md), [managing side effects](https://github.com/markerikson/react-redux-links/blob/master/redux-side-effects.md), [Redux architecture and best practices](https://github.com/markerikson/react-redux-links/blob/master/redux-architecture.md), and more.
* Our community has created thousands of Redux-related libraries, addons, and tools. The [**"Ecosystem" docs page**](https://redux.js.org/introduction/ecosystem) lists our recommendations, and there's a complete listing available in the [**Redux addons catalog**](https://github.com/markerikson/redux-ecosystem-links).
* If you're looking to learn from actual application codebases, the addons catalog also has a list of [**purpose-built examples and real-world applications**](https://github.com/markerikson/redux-ecosystem-links/blob/master/apps-and-examples.md).