express-handlebars

5.1.0 • Public • Published a month ago

Readme

Explore BETA

3 Dependencies

486 Dependents

27 Versions

Install

```
> npm i express-handlebars
```

Weekly Downloads

119,090

Version
5.1.0

License
BSD-3-Clause

Unpacked Size
83.1 kB

Total Files
27

Issues
3

Pull Requests
0

Homepage
github.com/express-handlebars/express-handlebars

Repository
github.com/express-handlebars/express-handlebars

Last publish
a month ago

Collaborators



Try on RunKit

Report a vulnerability

Express Handlebars

A Handlebars view engine for Express which doesn't suck.

npm v5.1.0 dependencies up to date

This package used to be named `express3-handlebars`. The previous `express-handlebars` package by @jneen can be found here.

Goals & Design

I created this project out of frustration with the existing Handlebars view engines for Express. As of version 3.x, Express got out of the business of being a generic view engine — this was a great decision — leaving developers to implement the concepts of layouts, partials, and doing file I/O for their template engines of choice.

Goals and Features

After building a half-dozen Express apps, I developed requirements and opinions about what a Handlebars view engine should provide and how it should be implemented. The following is that list:

- Add back the concept of "layout", which was removed in Express 3.x.
- Add back the concept of "partials" via Handlebars' partials mechanism.
- Support a directory of partials; e.g., `{{> foo/bar}}` which exists on the file system at `views/partials/foo/bar.handlebars`, by default.
- Smart file system I/O and template caching. When in development, templates are always loaded from disk. In production, raw files and compiled templates are cached, including partials.
- All async and non-blocking. File system I/O is slow and servers should not be blocked from handling requests while reading from disk. I/O queuing is used to avoid doing unnecessary work.
- Ability to easily precompile templates and partials for use on the client, enabling template sharing and reuse.
- Ability to use a different Handlebars module/implementation other than the Handlebars npm package.

Package Design

This package was designed to work great for both the simple and complex use cases. I *intentionally* made sure the full implementation is exposed and is easily overridable.

The package exports a function which can be invoked with no arguments or with a `config` object and it will return a function (closed over sensible defaults) which can be registered with an Express app. It's an engine factory function.

This exported engine factory has two properties which expose the underlying implementation:

- `ExpressHandlebars()` : The constructor function which holds the internal implementation on its `prototype` . This produces instance objects which store their configuration, `compiled` and `precompiled` templates, and expose an `engine()` function which can be registered with an Express app.
- `create()` : A convenience factory function for creating `ExpressHandlebars` instances.

An instance-based approach is used so that multiple `ExpressHandlebars` instances can be created with their own configuration, templates, partials, and helpers.

Installation

Install using npm:

```
$ npm install express-handlebars
```

Usage

This view engine uses sensible defaults that leverage the "Express-way" of structuring an app's views. This makes it trivial to use in basic apps:

Basic Usage

Directory Structure:

```
.
├── app.js
└── views
    ├── home.handlebars
    └── layouts
        └── main.handlebars
```

```
2 directories, 3 files
```

app.js:

Creates a super simple Express app which shows the basic way to register a Handlebars view engine using this package.

```
var express = require('express');
var exphbs  = require('express-handlebars');

var app = express();

app.engine('handlebars', exphbs());
app.set('view engine', 'handlebars');

app.get('/', function (req, res) {
    res.render('home');
});

app.listen(3000);
```

views/layouts/main.handlebars:

The main layout is the HTML page wrapper which can be reused for the different views of the app. `{{{body}}}` is used as a placeholder for where the main content should be rendered.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Example App</title>
</head>
<body>

    {{{body}}}
```

```
</body>
</html>
```

views/home.handlebars:

The content for the app's home view which will be rendered into the layout's `{{{body}}}`.

```
<h1>Example App: Home</h1>
```

Running the Example

The above example is bundled in this package's **examples directory**, where it can be run by:

```
$ cd examples/basic/
$ npm install
$ npm start
```

Using Instances

Another way to use this view engine is to create an instance(s) of `ExpressHandlebars`, allowing access to the full API:

```
var express = require('express');
var exphbs  = require('express-handlebars');

var app = express();
var hbs = exphbs.create({ /* config */ });

// Register `hbs.engine` with the Express app.
app.engine('handlebars', hbs.engine);
app.set('view engine', 'handlebars');

// ...still have a reference to `hbs`, on which methods like `getPa
// can be called.
```

Note: The **Advanced Usage** example demonstrates how `ExpressHandlebars` instances can be leveraged.

Template Caching

This view engine uses a smart template caching strategy. In development, templates will always be loaded from disk, i.e., no caching. In production, raw files and compiled Handlebars templates are aggressively cached.

The easiest way to control template/view caching is through Express' **view cache setting**:

```
app.enable('view cache');
```

Express enables this setting by default when in production mode, i.e.:

```
process.env.NODE_ENV === "production"
```

Note: All of the public API methods accept `options.cache`, which gives control over caching when calling these methods directly.

Layouts

A layout is simply a Handlebars template with a `{{{body}}}` placeholder. Usually it will be an HTML page wrapper into which views will be rendered.

This view engine adds back the concept of "layout", which was removed in Express 3.x. It can be configured with a path to the layouts directory, by default it's set to relative to `express settings.view + layouts/`

There are two ways to set a default layout: configuring the view engine's `defaultLayout` property, or setting **Express locals** `app.locals.layout`.

The layout into which a view should be rendered can be overridden per-request by assigning a different value to the `layout` request local. The following will render the "home" view with no layout:

```
app.get('/', function (req, res, next) {
    res.render('home', {layout: false});
});
```

Helpers

Helper functions, or "helpers" are functions that can be **registered with Handlebars** and can be called within a template. Helpers can be used for transforming output, iterating over data, etc. To keep with the spirit of *logic-less* templates, helpers are the place where logic should be defined.

Handlebars ships with some **built-in helpers**, such as: `with`, `if`, `each`, etc. Most application will need to extend this set of helpers to include app-specific logic and transformations. Beyond defining global helpers on `Handlebars`, this view engine supports `ExpressHandlebars` instance-level helpers via the `helpers` configuration property, and render-level helpers via `options.helpers` when calling the `render()` and `renderView()` methods.

The following example shows helpers being specified at each level:

app.js:

Creates a super simple Express app which shows the basic way to register `ExpressHandlebars` instance-level helpers, and override one at the render-level.

```
var express = require('express');
var exphbs  = require('express-handlebars');

var app = express();

var hbs = exphbs.create({
    // Specify helpers which are only registered on this instance.
    helpers: {
        foo: function () { return 'FOO!'; },
        bar: function () { return 'BAR!'; }
    }
});

app.engine('handlebars', hbs.engine);
app.set('view engine', 'handlebars');

app.get('/', function (req, res, next) {
    res.render('home', {
```

```
            showTitle: true,

            // Override `foo` helper only for this rendering.
            helpers: {
                foo: function () { return 'foo.'; }
            }
        });
    });

    app.listen(3000);
```

views/home.handlebars:

The app's home view which uses helper functions to help render the contents.

```
    <!DOCTYPE html>
    <html>
    <head>
        <meta charset="utf-8">
        <title>Example App - Home</title>
    </head>
    <body>

        <!-- Uses built-in `if` helper. -->
      {{#if showTitle}}
        <h1>Home</h1>
      {{/if}}

        <!-- Calls `foo` helper, overridden at render-level. -->
        <p>{{foo}}</p>

        <!-- Calls `bar` helper, defined at instance-level. -->
        <p>{{bar}}</p>

    </body>
    </html>
```

More on Helpers

Refer to the **Handlebars website** for more information on defining helpers:

- Expression Helpers
- Block Helpers

Metadata

Handlebars has a data channel feature that propagates data through all scopes, including helpers and partials. Values in the data channel can be accessed via the `{{@variable}}` syntax. Express Handlebars provides metadata about a template it renders on a `{{@exphbs}}` object allowing access to things like the view name passed to `res.render()` via `{{@exphbs.view}}`.

The following is the list of metadata that's accessible on the `{{@exphbs}}` data object:

- `cache` : Boolean whether or not the template is cached.
- `encoding` : String name of encoding for files.
- `view` : String name of the view passed to `res.render()`.
- `layout` : String name of the layout view.
- `data` : Original data object passed when rendering the template.
- `helpers` : Collection of helpers used when rendering the template.
- `partials` : Collection of partials used when rendering the template.
- `runtimeOptions` : Runtime Options used to render the template.

API

Configuration and Defaults

There are two main ways to use this package: via its engine factory function, or creating `ExpressHandlebars` instances; both use the same configuration properties and defaults.

```
var exphbs = require('express-handlebars');

// Using the engine factory:
exphbs({ /* config */ });

// Create an instance:
```

```
exphbs.create({ /* config */ });
```

The following is the list of configuration properties and their default values (if any):

handlebars=require('handlebars')

The Handlebars module/implementation. This allows for the `ExpressHandlebars` instance to use a different Handlebars module/implementation than that provided by the Handlebars npm package.

extname=".handlebars"

The string name of the file extension used by the templates. This value should correspond with the `extname` under which this view engine is registered with Express when calling `app.engine()`.

The following example sets up an Express app to use `.hbs` as the file extension for views:

```
var express = require('express');
var exphbs  = require('express-handlebars');

var app = express();

app.engine('.hbs', exphbs({extname: '.hbs'}));
app.set('view engine', '.hbs');
```

Note: Setting the app's `"view engine"` setting will make that value the default file extension used for looking up views.

encoding="utf8"

Default encoding when reading files.

layoutsDir

Default layouts directory is relative to `express settings.view + layouts/` The string path to the directory where the layout templates reside.

Note: If you configure Express to look for views in a custom location (e.g., `app.set('views', 'some/path/')`), and if your `layoutsDir` is not relative to

`express settings.view + layouts/`, you will need to reflect that by passing an updated path as the `layoutsDir` property in your configuration.

partialsDir

Default partials directory is relative to `express settings.view + partials/` The string path to the directory where the partials templates reside or object with the following properties:

- `dir` : The string path to the directory where the partials templates reside.
- `namespace` : Optional string namespace to prefix the partial names.
- `templates` : Optional collection (or promise of a collection) of templates in the form: `{filename: template}`.

Note: If you configure Express to look for views in a custom location (e.g., `app.set('views', 'some/path/')`), and if your `partialsDir` is not relative to `express settings.view + partials/`, you will need to reflect that by passing an updated path as the `partialsDir` property in your configuration.

Note: Multiple partials dirs can be used by making `partialsDir` an array of strings, and/or config objects as described above. The namespacing feature is useful if multiple partials dirs are used and their file paths might clash.

defaultLayout

The string name or path of a template in the `layoutsDir` to use as the default layout. `main` is used as the default. This is overridden by a `layout` specified in the app or response `locals`. **Note:** A falsy value will render without a layout; e.g., `res.render('home', {layout: false});`.

helpers

An object which holds the helper functions used when rendering templates with this `ExpressHandlebars` instance. When rendering a template, a collection of helpers will be generated by merging: `handlebars.helpers` (global), `helpers` (instance), and `options.helpers` (render-level). This allows Handlebars' `registerHelper()` function to operate as expected, will providing two extra levels over helper overrides.

compilerOptions

An object which holds options that will be passed along to the Handlebars compiler functions: `Handlebars.compile()` and `Handlebars.precompile()`.

`runtimeOptions`

An object which holds options that will be passed along to the template function in addition to the `data`, `helpers`, and `partials` options. See **Runtime Options** for a list of available options.

Properties

The public API properties are provided via `ExpressHandlebars` instances. In additional to the properties listed in the **Configuration and Defaults** section, the following are additional public properties:

`engine`

A function reference to the `renderView()` method which is bound to `this` `ExpressHandlebars` instance. This bound function should be used when registering this view engine with an Express app.

`extname`

The normalized `extname` which will *always* start with `.` and defaults to `.handlebars`.

`compiled`

An object cache which holds compiled Handlebars template functions in the format: `{"path/to/template": [Function]}`.

`precompiled`

An object cache which holds precompiled Handlebars template strings in the format: `{"path/to/template": [String]}`.

Methods

The following is the list of public API methods provided via `ExpressHandlebars` instances:

Note: All of the public methods return a `Promise` (with the exception of `renderView()` which is the interface with Express.)

`getPartials([options])`

Retrieves the partials in the `partialsDir` and returns a Promise for an object mapping the partials in the form `{name: partial}`.

By default each partial will be a compiled Handlebars template function. Use `options.precompiled` to receive the partials as precompiled templates — this is useful for sharing templates with client code.

Parameters:

- `[options]` : Optional object containing any of the following properties:
  - `[cache]` : Whether cached templates can be used if they have already been requested. This is recommended for production to avoid unnecessary file I/O.
  - `[encoding]` : File encoding.
  - `[precompiled=false]` : Whether precompiled templates should be provided, instead of compiled Handlebars template functions.

The name of each partial corresponds to its location in `partialsDir` . For example, consider the following directory structure:

```
views
└── partials
    ├── foo
    │   └── bar.handlebars
    └── title.handlebars

2 directories, 2 files
```

`getPartials()` would produce the following result:

```
var hbs = require('express-handlebars').create();

hbs.getPartials().then(function (partials) {
    console.log(partials);
    // => { 'foo/bar': [Function],
    // =>    title: [Function] }
});
```

getTemplate(filePath, [options])

Retrieves the template at the specified `filePath` and returns a Promise for the compiled Handlebars template function.

Use `options.precompiled` to receive a precompiled Handlebars template.

Parameters:

- `filePath` : String path to the Handlebars template file.
- `[options]` : Optional object containing any of the following properties:
  - `[cache]` : Whether a cached template can be used if it have already been requested. This is recommended for production to avoid necessary file I/O.
  - `[encoding]` : File encoding.
  - `[precompiled=false]` : Whether a precompiled template should be provided, instead of a compiled Handlebars template function.

`getTemplates(dirPath, [options])`

Retrieves all the templates in the specified `dirPath` and returns a Promise for an object mapping the compiled templates in the form `{filename: template}` .

Use `options.precompiled` to receive precompiled Handlebars templates — this is useful for sharing templates with client code.

Parameters:

- `dirPath` : String path to the directory containing Handlebars template files.
- `[options]` : Optional object containing any of the following properties:
  - `[cache]` : Whether cached templates can be used if it have already been requested. This is recommended for production to avoid necessary file I/O.
  - `[encoding]` : File encoding.
  - `[precompiled=false]` : Whether precompiled templates should be provided, instead of a compiled Handlebars template function.

`render(filePath, context, [options])`

Renders the template at the specified `filePath` with the `context`, using this instance's `helpers` and partials by default, and returns a Promise for the resulting string.

Parameters:

- `filePath`: String path to the Handlebars template file.
- `context`: Object in which the template will be executed. This contains all of the values to fill into the template.
- `[options]`: Optional object which can contain any of the following properties which affect this view engine's behavior:
  - `[cache]`: Whether a cached template can be used if it have already been requested. This is recommended for production to avoid unnecessary file I/O.
  - `[encoding]`: File encoding.
  - `[data]`: Optional object which can contain any data that Handlebars will pipe through the template, all helpers, and all partials. This is a side data channel.
  - `[helpers]`: Render-level helpers that will be used instead of any instance-level helpers; these will be merged with (and will override) any global Handlebars helper functions.
  - `[partials]`: Render-level partials that will be used instead of any instance-level partials. This is used internally as an optimization to avoid re-loading all the partials.
  - `[runtimeOptions]`: Optional object which can contain options passed to the template function.

`renderView(viewPath, options|callback, [callback])`

Renders the template at the specified `viewPath` as the `{{{body}}}` within the layout specified by the `defaultLayout` or `options.layout`. Rendering will use this instance's `helpers` and partials, and passes the resulting string to the `callback`.

This method is called by Express and is the main entry point into this Express view engine implementation. It adds the concept of a "layout" and delegates rendering to the `render()` method.

The `options` will be used both as the context in which the Handlebars templates are rendered, and to signal this view engine on how it should behave, e.g., `options.cache=false` will *always* load the templates from disk.

Parameters:

- `viewPath` : String path to the Handlebars template file which should serve as the `{{{body}}}` when using a layout.
- `[options]` : Optional object which will serve as the context in which the Handlebars templates are rendered. It may also contain any of the following properties which affect this view engine's behavior:
    - `[cache]` : Whether cached templates can be used if they have already been requested. This is recommended for production to avoid unnecessary file I/O.
    - `[encoding]` : File encoding.
    - `[data]` : Optional object which can contain any data that Handlebars will pipe through the template, all helpers, and all partials. This is a side data channel.
    - `[helpers]` : Render-level helpers that will be merged with (and will override) instance and global helper functions.
    - `[partials]` : Render-level partials will be merged with (and will override) instance and global partials. This should be a `{partialName: fn}` hash or a Promise of an object with this shape.
    - `[layout]` : Optional string path to the Handlebars template file to be used as the "layout". This overrides any `defaultLayout` value. Passing a falsy value will render with no layout (even if a `defaultLayout` is defined).
    - `[runtimeOptions]` : Optional object which can contain options passed to the template function.
- `callback` : Function to call once the template is retrieved.

Hooks

The following is the list of protected methods that are called internally and serve as *hooks* to override functionality of `ExpressHandlebars` instances. A value or a promise can be returned from these methods which allows them to perform async operations.

`_compileTemplate(template, options)`

This hook will be called when a Handlebars template needs to be compiled. This function needs to return a compiled Handlebars template function, or a promise for one.

By default this hook calls `Handlebars.compile()`, but it can be overridden to preform operations before and/or after Handlebars compiles the template. This is useful if you wanted to first process Markdown within a Handlebars template.

Parameters:

- `template`: String Handlebars template that needs to be compiled.
- `options`: Object `compilerOptions` that were specified when the `ExpressHandlebars` instance as created. This object should be passed along to the `Handlebars.compile()` function.

`_precompileTemplate(template, options)`

This hook will be called when a Handlebars template needs to be precompiled. This function needs to return a serialized Handlebars template spec. string, or a promise for one.

By default this hook calls `Handlebars.precompile()`, but it can be overridden to preform operations before and/or after Handlebars precompiles the template. This is useful if you wanted to first process Markdown within a Handlebars template.

Parameters:

- `template`: String Handlebars template that needs to be precompiled.
- `options`: Object `compilerOptions` that were specified when the `ExpressHandlebars` instance as created. This object should be passed along to the `Handlebars.compile()` function.

`_renderTemplate(template, context, options)`

This hook will be called when a compiled Handlebars template needs to be rendered. This function needs to returned the rendered output string, or a promise for one.

By default this hook simply calls the passed-in `template` with the `context` and `options` arguments, but it can be overridden to perform operations before and/or after

rendering the template.

Parameters:

- `template` : Compiled Handlebars template function to call.
- `context` : The context object in which to render the `template` .
- `options` : Object that contains options and metadata for rendering the template:
  - `data` : Object to define custom `@variable` private variables.
  - `helpers` : Object to provide custom helpers in addition to the globally defined helpers.
  - `partials` : Object to provide custom partials in addition to the globally defined partials.
  - `...runtimeOptions` : Other options specified by the `runtimeOptions` value.

Examples

Basic Usage

This example shows the most basic way to use this view engine.

Advanced Usage

This example is more comprehensive and shows how to use many of the features of this view engine, including helpers, partials, multiple layouts, etc.

As noted in the **Package Design** section, this view engine's implementation is instance-based, and more advanced usages can take advantage of this. The Advanced Usage example demonstrates how to use an `ExpressHandlebars` instance to share templates with the client, among other features.

License

Keywords

express express3 handlebars view layout partials templates

Support

Help

Community

Advisories

Status

Contact npm

Company

About

Blog

Press

Terms & Policies

Policies

Terms of Use

Code of Conduct

Privacy