Todo App

Todo apps are considered good tlearning examples because they have many components that you will see in a typical, real-life project. Also, this sort of app is popular when it comes to showcasing browser JavaScript frameworks.

Just look at the famous TodoMVC project (http://todomvc.com), which has a few dozen Todo apps for the various front-end JavaScript frameworks.

In our Todo app, we'll use MongoDB, Mongoskin, Jade, web forms, Less style sheets, and cross-site request forgery (CSRF) protection. We'll intentionally not use Backbone.js or AngularJS, because the goal is to demonstrate how to build traditional web sites with the use of forms, redirects, and server-side template rendering. We'll also look at how to plug in CSRF and Less. As a bonus, there will be a few AJAX/XHR calls to RESTful API-ish endpoints, because it's hard to build a modern user interface/experience without such calls. You should know how to use them in this hybrid web site architecture (traditional server-side HTML rendering with some AJAX/XHR calls).

This project may seem complex, so before you start coding it, here's an overview of how this tutorial presents the steps to achieve the end product:

- Overview
- Setup
- App.js

• Routes

• Jade

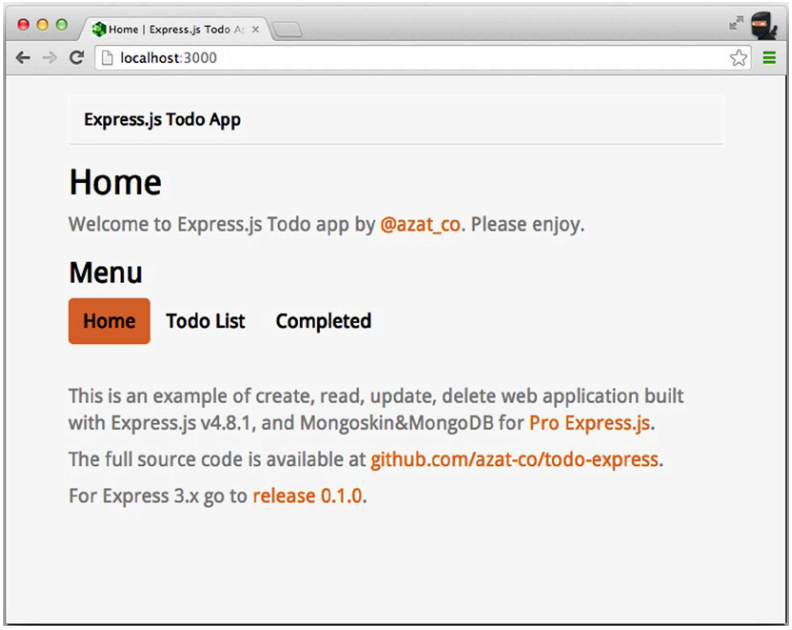
• Less

To preview what we are going to achieve, let's start with some screenshots of the Todo app showing how the user interface functions. The image below shows the Home page, which has a heading, a menu, and some introductory text. The menu consists of three items:

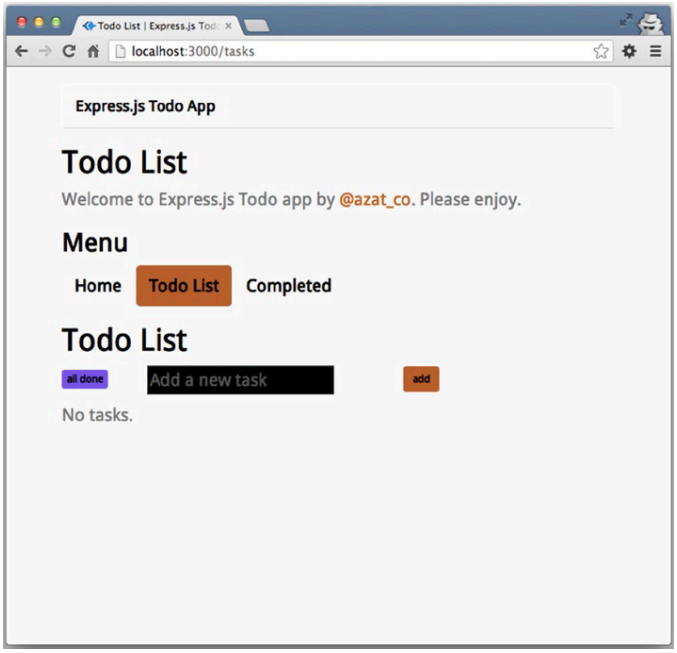
• Home: The currently displayed page

• Todo List: A list of tasks to do

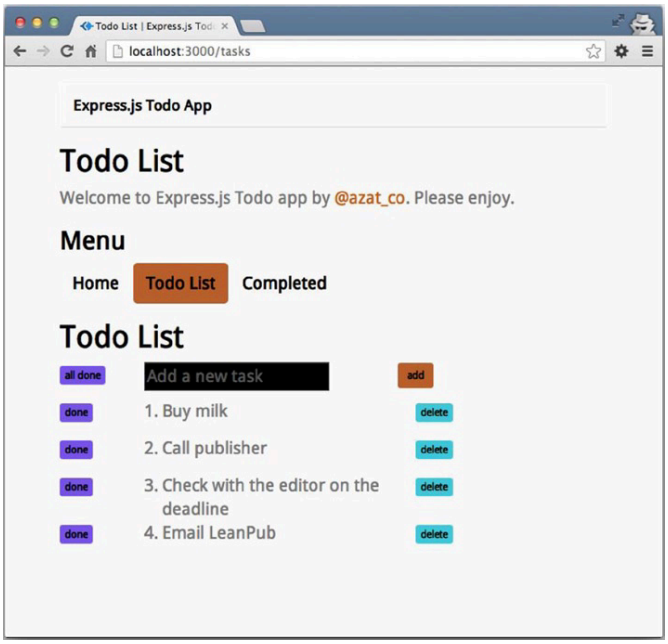
• Completed: A list of completed tasks



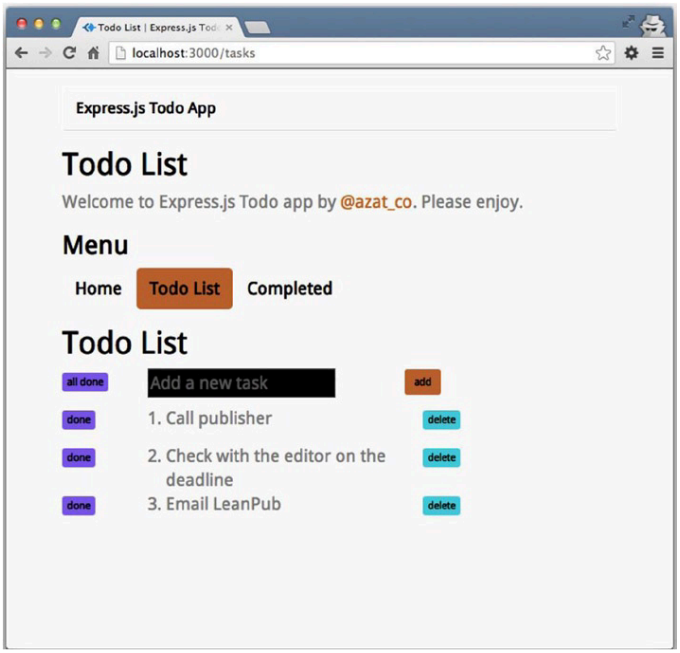
On the Todo List page, there's an empty list, as shown in image below. There's also an entry form for the new task and an "add" button.



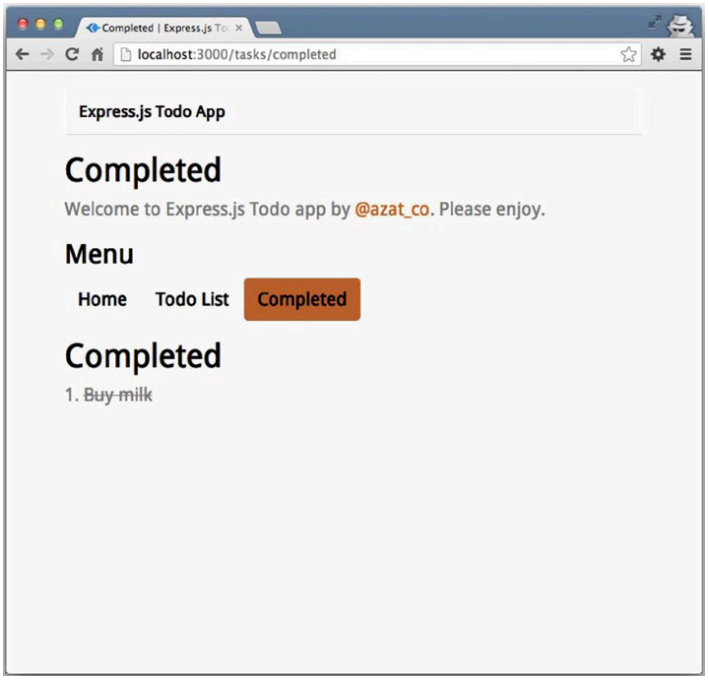
The image below shows the result of adding four items to the Todo List. Each task has a “done” button to its left and a “delete” button to its right, the functions of which are exactly as you might guess. They mark the task as completed (i.e., move it to the Completed page) and remove the task, respectively.



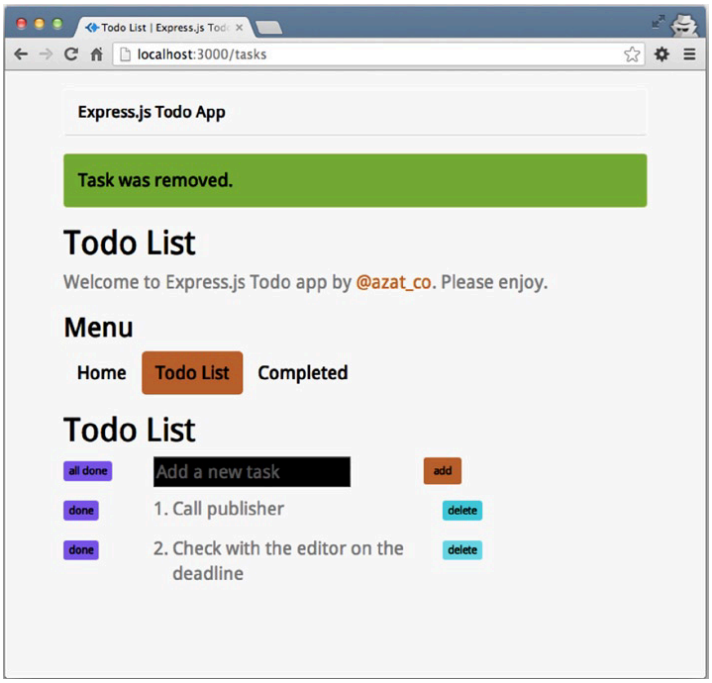
The next image below the result of clicking the “done” button for the “Buy milk” task. The item has disappeared from the Todo List and the list has been renumbered.



However, the completed “Buy milk” task has not disappeared from the app altogether. It is now in the Completed list on the Completed page, as shown below.



Deletion of an item from the Todo List page after the “delete” button is clicked is the action performed via an AJAX/XHR request. The image below shows the purple-highlighted notification message that appears when a task is deleted (in this case, the “Email LeanPub” task). The rest of the logic is implemented via GETs and POSTs (by forms).



Setup

We begin the setup of the Todo app by creating a new folder:

```
$ mkdir todo-express

$ cd todo-express
```

As usual, we start by taking care of the dependencies. This command gives us the basic package.json file:

```
$ npm init
```

We need to add the following extra dependencies to package.json:

• express v4.8.1: For Express.js framework

• body-parser v1.6.6: For processing payloads

• cookie-parser v1.3.2: For processing cookies and for sessions

• express-session v1.7.6: For session support

• csurf v1.5.0: For CSRF security

• errorhandler v1.1.1: For basic error handling

• jade v1.5.0: For Jade template

• less-middleware v1.0.4: For Less support

• method-override v2.1.3: For clients that don’t support all HTTP methods

• mongoskin v1.4.4: For MongoDB connections

• morgan v1.2.3: For logging of requests

• serve-favicon v2.1.1: For favicon support

One of the ways to add the preceding list of dependencies is to utilize the --save (-s) option of npm install:

```
$ npm install less-middleware@1.0.4 --save
$ npm install mongoskin@1.4.4 --save
```

Another way is to add entries to package.json and run $ npm install:

```
{
"name": "todo-express",
"version": "0.2.0",
```

```
"private": true,
"scripts": {
"start": "node app.js"
},
"dependencies": {
"body-parser": "1.6.6",
"cookie-parser": "1.3.2",
"csurf": "1.5.0",
"errorhandler": "1.1.1",
"express": "4.8.1",
"express-session": "1.7.6",
"jade": "1.5.0",
"less-middleware": "1.0.4",
"method-override": "2.1.3",
"mongoskin": "1.4.4",
"morgan": "1.2.3",
"serve-favicon": "2.1.1"
}
}
```

Now, install the MongoDB database if you don’t have it installed already. The database is not the same as NPM modules mongodb and mongoskin, which are drivers. These libraries allow us to interact with the MongoDB database, but we still need both the driver and the database.

On OS X, you can use brew to install MongoDB (or upgrade to v2.6.3):

```
$ brew update
$ brew install mongodb
$ mongo --version
```

For more flavors of MongoDB installations, check out the official docs1 and/or Practical Node.js. The end version of the app (0.20.0) has the following folder and file structure.

```
/todo-express
        /public
                /bootstrap
                        *.less
                /images
                /javascripts
                        main.js
                        jquery.js
                /stylesheets
                        style.css
                        main.less
        favicon.ico
        /routes
                tasks.js
                index.js
```

```
        /views
                tasks_completed.jade
                layout.jade
                index.jade
                tasks.jade
        app.js
        readme.md
        package.json
```

The *.less in the bootstrap folder means there are a bunch of Bootstrap (the CSS framework, http://getbootstrap.com/) source files.

App.js

This section presents a breakdown of the Express.js-generated app.js file with the addition of routes, database, session, Less, and app.param() middleware.

First, we import dependencies with the Node.js global require() function:

```
var express = require('express');
```

Similarly, we get access to our own modules, which are the app's routes:

```
var routes = require('./routes');
var tasks = require('./routes/tasks');
```

We need the core http and path modules as well:

```
var http = require('http');
var path = require('path');
```

Mongoskin is a better alternative to the native MongoDB driver because it provides additional features and methods:

```
var mongoskin = require('mongoskin');
```

One line is all we need to get the database connection object. The first parameter follows the standard URI convention of protocol://username:password@host:port/database:

```
var db = mongoskin.db('mongodb://localhost:27017/todo?auto_reconnect', {safe:true});
```

We set up the app itself:

```
var app = express();
```

Now, we import the middleware dependencies from NPM modules:

```
var favicon = require('serve-favicon'),
logger = require('morgan'),
bodyParser = require('body-parser'),
methodOverride = require('method-override'),
cookieParser = require('cookie-parser'),
session = require('express-session'),
csrf = require('csurf'),
```

```
errorHandler = require('errorhandler');
```

In this middleware, we export the database object to all middleware functions. By doing so, we'll be able to perform database operations in the routes modules:

```
app.use(function(req, res, next) {
req.db = {};
```

We simply store the tasks collection in every request:

```
req.db.tasks = db.collection('tasks');
next();
})
```

This line allows us to access appname from within every Jade template:

```
app.locals.appname = 'Express.js Todo App'
```

We set the server port, either to the environment variable or, if that's undefined, to 3000:

```
app.set('port', process.env.PORT || 3000);
```

These statements tell Express.js where templates live and what file extension to prepend in case the extension is omitted during the render calls:

```
app.set('views', __dirname + '/views');
app.set('view engine', 'jade');
```

The following displays the Express.js favicon (the graphic in the URL address bar of browsers):

```
app.use(favicon(path.join('public','favicon.ico')));
```

The out-of-the-box logger will print requests in the terminal window:

```
app.use(logger('dev'));
```

The bodyParser() middleware is needed to painlessly access incoming data:

```
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended: true}));
```

The methodOverride() middleware is a workaround for HTTP methods that involve headers. It's not essential for this example, but we'll leave it here:

```
app.use(methodOverride());
```

To use CSRF, we need cookieParser() and session(). The following, weird-looking strings are secrets. You want them to be random and to come from environment variables (process.env), not hard-coded.

```
app.use(cookieParser('CEAF3FA4-F385-49AA-8FE4-54766A9874F1'));
app.use(session({
secret: '59B93087-78BC-4EB9-993A-A61FC844F6C9',
resave: true,
saveUninitialized: true
}));
```

The express-session options are covered in Chapter 3, but, as shown in the preceding code, v1.7.6 (which we use here) has a resave option, which saves unmodified sessions if set to true, and a saveUninitialized option, which saves a new, but unmodified session, if set to true. The default value for both options is true. The recommended values are false for resave and true for saveUninitialized. If you don't specify values for these options, then you'll get warnings, because these options' defaults will likely change in the future. So, it's good to explicitly set the options. Alternatively, to suppress these warnings, you can use the environment variable:

```
$ NO_DEPRECATION=express-session node app
```

Next, we apply the csrf() middleware itself. The order is important: csrf() must be preceded by cookieParser() and session().

```
app.use(csrf());
```

To process Less style sheets into CSS ones, we utilize less-middleware in this manner:

```
app.use(require('less-middleware')(path.join(__dirname, 'public')));
```

The other static files are also in the public folder:

```
app.use(express.static(path.join(__dirname, 'public')));
```

Remember CSRF? The main trick here is to use req.csrfToken(), which is created by the middleware that we previously applied in app.js. This is how we expose the CSRF token to templates:

```
app.use(function(req, res, next) {
```

```
res.locals._csrf = req.csrfToken();

return next();

})
```

When there's a request that matches route/RegExp with :task_id in it, this block is executed:

```
app.param('task_id', function(req, res, next, taskId) {
```

The value of task ID is in taskId, and we query the database to find that object:

```
req.db.tasks.findById(taskId, function(error, task){
```

It's tremendously important to check for errors and empty results:

```
if (error) return next(error);

if (!task) return next(new Error('Task is not found.'));
```

If there's data, we store it in the request and proceed to the next middleware:

```
req.task = task;

return next();

});

});
```

Now it's time to define our routes. We start with the Home page:

```
app.get('/', routes.index);
```

Next is the Todo List page:

```
app.get('/tasks', tasks.list);
```

The following route will mark all tasks in the Todo List as completed if the user clicks the “all done” button. In a REST API, the HTTP method would be PUT, but, because we are building classical web apps with forms, we have to use POST:

```
app.post('/tasks', tasks.markAllCompleted)
```

The same URL for adding new tasks is used to mark all tasks completed, but, in the previous method (markAllCompleted()), you’ll see how we handle flow control:

```
app.post('/tasks', tasks.add);
```

To mark a single task completed, we use the aforementioned :task_id string in our URL pattern (in a REST API, this would be a PUT request):

```
app.post('/tasks/:task_id', tasks.markCompleted);
```

Unlike with the previous POST route, we utilize Express.js param middleware with a :task_id token:

```
app.del('/tasks/:task_id', tasks.del);
```

For our Completed page, we define this route:

```
app.get('/tasks/completed', tasks.completed);
```

In case of malicious attacks or mistyped URLs, catching all requests with * is a user-friendly activity. Keep in mind that, if we had a match previously, Node.js won't come to execute this block.

```
app.all('*', function(req, res){
res.status(404).send();
})
```

It's possible to configure different behavior based on environments:

```
if ('development' == app.get('env')) {
app.use(errorHandler());
}
```

Finally, we spin up our application with the good old http method:

```
http.createServer(app).listen(app.get('port'),
function(){
console.log('Express server listening on port '
+ app.get('port'));
}
);
```

The full content of the app.js file follows:

```
var express = require('express');
var routes = require('./routes');
```

```
var tasks = require('./routes/tasks');
var http = require('http');
var path = require('path');
var mongoskin = require('mongoskin');
var db = mongoskin.db('mongodb://localhost:27017/todo?auto_reconnect', {safe:true});
var app = express();
var favicon = require('serve-favicon'),
logger = require('morgan'),
bodyParser = require('body-parser'),
methodOverride = require('method-override'),
cookieParser = require('cookie-parser'),
session = require('express-session'),
csrf = require('csurf'),
errorHandler = require('errorhandler');
app.use(function(req, res, next) {
req.db = {};
req.db.tasks = db.collection('tasks');
next();
})
app.locals.appname = 'Express.js Todo App'
app.set('port', process.env.PORT || 3000);
app.set('views', __dirname + '/views');
app.set('view engine', 'jade');
```

```
app.use(favicon(path.join('public','favicon.ico')));
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended: true}));
app.use(methodOverride());
app.use(cookieParser('CEAF3FA4-F385-49AA-8FE4-54766A9874F1'));
app.use(session({
secret: '59B93087-78BC-4EB9-993A-A61FC844F6C9',
resave: true,
saveUninitialized: true
}));
app.use(csrf());
app.use(require('less-middleware')(path.join(__dirname, 'public')));
app.use(express.static(path.join(__dirname, 'public')));
app.use(function(req, res, next) {
res.locals._csrf = req.csrfToken();
return next();
})
app.param('task_id', function(req, res, next, taskId) {
req.db.tasks.findById(taskId, function(error, task){
if (error) return next(error);
if (!task) return next(new Error('Task is not found.'));
req.task = task;
```

```
return next();
});
});
app.get('/', routes.index);
app.get('/tasks', tasks.list);
app.post('/tasks', tasks.markAllCompleted)
app.post('/tasks', tasks.add);
app.post('/tasks/:task_id', tasks.markCompleted);
app.delete('/tasks/:task_id', tasks.del);
app.get('/tasks/completed', tasks.completed);
app.all('*', function(req, res){
res.status(404).send();
})
// development only
if ('development' == app.get('env')) {
app.use(errorHandler());
}
http.createServer(app).listen(app.get('port'), function(){
console.log('Express server listening on port ' + app.get('port'));
});
```

Routes

There are only two files in the routes folder. One of them, routes/index.js, serves the home page (e.g., http://localhost:3000/) and is straightforward:

```
exports.index = function(req, res){
res.render('index', { title: 'Home' });
};
```

The remaining logic that deals with tasks has been placed in todo-express/routes/tasks.js. Let's break down that file a bit further. We start by exporting a list() request handler that gives us a list of incomplete tasks:

```
exports.list = function(req, res, next){
```

To do so, we perform a database search with a completed=false query:

```
req.db.tasks.find({
completed: false
}).toArray(function(error, tasks){
```

In the callback, we need to check for any errors:

```
if (error) return next(error);
```

Because we use toArray(), we can send the data directly to the template:

```
res.render('tasks', {
title: 'Todo List',
tasks: tasks || []
```

```
});

});

};
```

Adding a new task requires us to check for the name parameter:

```
exports.add = function(req, res, next){

if (!req.body || !req.body.name)

return next(new Error('No data provided.'));
```

Thanks to our middleware, we already have a database collection in the req object and the default value for the task is incomplete (completed: false):

```
req.db.tasks.save({

name: req.body.name,

completed: false

}, function(error, task){
```

Again, it’s important to check for errors and propagate them with the Express.js next() function:

```
if (error) return next(error);

if (!task) return next(new Error('Failed to save.'));
```

Logging is optional. However, it’s useful for learning and debugging:

```
console.info('Added %s with id=%s', task.name, task._id);
```

Lastly, we redirect back to the Todo List page when the saving operation has finished successfully:

```
res.redirect('/tasks');
})
};
```

This method marks all incomplete tasks as complete:

```
exports.markAllCompleted = function(req, res, next) {
```

Because we had to reuse the POST route, and because it's a good illustration of flow control, we check for the all_done parameter to determine whether this request comes from the "all done" button or the "add" button:

```
if (!req.body.all_done
|| req.body.all_done !== 'true')
return next();
```

If the execution has come this far, we perform a database query with the multi: true option (update manydocuments). This query will assign the completed property to true on all unfinished tasks (completed: false) with the $set directive.

```
req.db.tasks.update({
completed: false
}, {$set: {
completed: true
```

```
}}, {multi: true}, function(error, count){
```

Next, we perform significant error handling, logging, and redirection back to Todo List page:

```
if (error) return next(error);
console.info('Marked %s task(s) completed.', count);
res.redirect('/tasks');
})
};
```

The Completed route is similar to the Todo List route, except for the completed flag value, which is true in this case:

```
exports.completed = function(req, res, next) {
req.db.tasks.find({
completed: true
}).toArray(function(error, tasks) {
res.render('tasks_completed', {
title: 'Completed',
tasks: tasks || []
});
});
};
```

This is the route that takes care of marking a single task as done. We use updateById, but we could accomplish the same thing with a plain update() method from the Mongoskin/MongoDB API.

On the $set line, instead of the req.body.completed value, we use the expression completed:

```
req.body.
completed === 'true'. It is needed because the incoming value of req.body.completed is a string and not a boolean.
exports.markCompleted = function(req, res, next) {
if (!req.body.completed)
return next(new Error('Param is missing.'));
req.db.tasks.updateById(req.task._id, {
$set: {completed: req.body.completed === 'true'}},
function(error, count) {
```

Once more, we perform error and results checks: (update() and updateById() don't return an object, but, instead, return the count of the affected documents):

```
if (error) return next(error);
if (count !==1)
return next(new Error('Something went wrong.'));
console.info('Marked task %s with id=%s completed.',
req.task.name,
req.task._id);
```

```
res.redirect('/tasks');
}
)
}
```

Delete is the single route called by an AJAX request. However, there's nothing special about its implementation. The only difference is that we don't redirect, but send status 200 back. Alternatively the remove() method can be used instead of removeById().

```
exports.del = function(req, res, next) {
req.db.tasks.removeById(req.task._id, function(error, count) {
if (error) return next(error);
if (count !==1) return next(new Error('Something went wrong.'));
console.info('Deleted task %s with id=%s completed.',
req.task.name,
req.task._id);
res.status(204).send();
});
}
```

For your convenience, here's the full content of the todo-express/routes/tasks.js file:

```
exports.list = function(req, res, next){
req.db.tasks.find({completed: false}).toArray(function(error, tasks){
if (error) return next(error);
```

```
res.render('tasks', {
title: 'Todo List',
tasks: tasks || []
});
});
};
exports.add = function(req, res, next){
if (!req.body || !req.body.name) return next(new Error('No data provided.'));
req.db.tasks.save({
name: req.body.name,
completed: false
}, function(error, task){
if (error) return next(error);
if (!task) return next(new Error('Failed to save.'));
console.info('Added %s with id=%s', task.name, task._id);
res.redirect('/tasks');
})
};
exports.markAllCompleted = function(req, res, next) {
if (!req.body.all_done || req.body.all_done !== 'true') return next();
req.db.tasks.update({
completed: false
}, {$set: {
```

```
completed: true
}}, {multi: true}, function(error, count){
if (error) return next(error);
console.info('Marked %s task(s) completed.', count);
res.redirect('/tasks');
})
};
exports.completed = function(req, res, next) {
req.db.tasks.find({completed: true}).toArray(function(error, tasks) {
res.render('tasks_completed', {
title: 'Completed',
tasks: tasks || []
});
});
};
exports.markCompleted = function(req, res, next) {
if (!req.body.completed) return next(new Error('Param is missing.'));
req.db.tasks.updateById(req.task._id, {$set: {completed: req.body.completed === 'true'}},
function(error, count) {
if (error) return next(error);
if (count !==1) return next(new Error('Something went wrong.'));
console.info('Marked task %s with id=%s completed.', req.task.name, req.task._id);
res.redirect('/tasks');
```

```
})
};
exports.del = function(req, res, next) {
req.db.tasks.removeById(req.task._id, function(error, count) {
if (error) return next(error);
if (count !==1) return next(new Error('Something went wrong.'));
console.info('Deleted task %s with id=%s completed.', req.task.name, req.task._id);
res.status(204).send();
});
};
```

So far we've implemented the main server file app.js and its routes that perform different database operations.

Now we can proceed to the templates.

Jade

In the Todo app, we use four templates:

• layout.jade: The skeleton of HTML pages that is used on all pages

• index.jade: Home page

• tasks.jade: Todo List page

• tasks_completed.jade: Completed page

Let's go through each file, starting with layout.jade. It starts with doctype, html, and head types:

```
doctype html
```

```
html
    head
```

We should set the appname variable:

```
title= title + ' | ' + appname
```

Next, we include *.css files and Express.js will serve their contents from Less files:

```
link(rel="stylesheet", href="/stylesheets/style.css")
link(rel="stylesheet", href="/bootstrap/bootstrap.css")
link(rel="stylesheet", href="/stylesheets/main.css")
```

The body with Bootstrap structure consists of the .container and .navbar classes. To read more about these and other classes, go to http://getbootstrap.com/css/.

```
body
    .container
        .navbar.navbar-default
            .container
                .navbar-header
                    a.navbar-brand(href='/')= appname
    .alert.alert-dismissable
    h1= title
    p Welcome to Express.js Todo app by 
        a(href='http://twitter.com/azat_co') @azat_co
```

```
|. Please enjoy.
```

This is the place where other jade templates (like tasks.jade) will be imported:

```
block content
```

The last lines include front-end JavaScript files:

```
script(src='/javascripts/jquery.js', type="text/javascript")

script(src='/javascripts/main.js', type="text/javascript")
```

The following is the full layout.jade file:

```
doctype html

html

        head

                title= title + ' | ' + appname

                link(rel="stylesheet", href="/stylesheets/style.css")

                link(rel="stylesheet", href="/bootstrap/bootstrap.css")

                link(rel="stylesheet", href="/stylesheets/main.css")

body

        .container

                .navbar.navbar-default

.                       container

                                .navbar-header

                                        a.navbar-brand(href='/')= appname
```

```
        .alert.alert-dismissable
        h1= title
        p Welcome to Express.js Todo app by&amp;nbsp;
                a(href='http://twitter.com/azat_co') @azat_co
                |. Please enjoy.
        block content
script(src='/javascripts/jquery.js', type="text/javascript")
script(src='/javascripts/main.js', type="text/javascript")
```

The index.jade file is our Home page and is quite vanilla. Its most interesting component is the nav-pills menu, which is a Bootstrap class for tabbed navigation. The rest of the file is just static hypertext:

```
extends layout
block content
        .menu
                h2 Menu
                ul.nav.nav-pills
                        li.active
                                a(href="/tasks") Home
                        li
                                a(href="/tasks") Todo List
                        li
                                a(href="/tasks") Completed
```

```
        .home
                p This is an example of create, read, update, delete web application built with
                Express.js v4.8.1, and Mongoskin&MongoDB for 
                        a(href="http://proexpressjs.com") Pro Express.js
                        |.
                p The full source code is available at 
                        a(href='http://github.com/azat-co/todo-express') github.com/azat-co/todo-
                        express
                        |.
                p For Express 3.x go to 
                        a(href="https://github.com/azat-co/todo-express/releases/tag/v0.1.0")
                        release 0.1.0
                        |.
```

Next is tasks.jade, which uses extends layout:

```
extends layout

block content
```

Next is our main page of specific content:

```
.menu
  h2 Menu
  ul.nav.nav-pills
    li
      a(href='/') Home
    li.active
      a(href='/tasks') Todo List
    li
       a(href="/tasks/completed") Completed
```

```
h1= title
```

The div with the list class will hold the Todo List:

```
.list
  .item.add-task
```

The form to mark all items as done has a CSRF token (locals._csrf) in a hidden field and uses the POST method pointed to /tasks:

```
div.action

form(action='/tasks', method='post')

input(type='hidden', value='true', name='all_done')

input(type='hidden', value=locals._csrf, name='_csrf')

input(type='submit', class='btn btn-success btn-xs', value='all done')
```

A similar CSRF-enabled form is used for new task creation:

```
form(action='/tasks', method='post')

input(type='hidden', value=locals._csrf, name='_csrf')

div.name

input(type='text', name='name', placeholder='Add a new task')

div.delete

input.btn.btn-primary.btn-sm(type='submit', value='add')
```

When we start the app for the first time (or clean the database), there are no tasks:

```
if (tasks.length === 0)

| No tasks.
```

Jade supports iterations with the each command:

```
each task, index in tasks
  .item
    div.action
```

This form submits data to its individual task route:

```
form(action='/tasks/#{task._id}', method='post')
input(type='hidden', value=task._id.toString(), name='id')
input(type='hidden', value='true', name='completed')
input(type='hidden', value=locals._csrf, name='_csrf')
input(type='submit', class='btn btn-success btn-xs task-done', value='done')
```

The index variable is used to display order in the list of tasks:

```
div.num
  span=index+1
    |. 
div.name
  span.name=task.name
  //- no support for DELETE method in forms
  //- http://amundsen.com/examples/put-delete-forms/
  //- so do XHR request instead from public/javascripts/main.js
```

The “delete” button doesn’t have anything fancy attached to it, because events are attached to these buttons from the main.js front-end JavaScript file:

```
div.delete
```

```
  a(class='btn btn-danger btn-xs task-delete', data-task-id=task._id.toString(),
data-csrf=locals._csrf) delete
```

The full source code of tasks.jade is provided here:

```
extends layout
block content
.menu
h2 Menu
ul.nav.nav-pills
li
a(href='/') Home
li.active
a(href='/tasks') Todo List
li
a(href="/tasks/completed") Completed
h1= title
.list
.item.add-task
div.action
form(action='/tasks', method='post')
input(type='hidden', value='true', name='all_done')
input(type='hidden', value=locals._csrf, name='_csrf')
input(type='submit', class='btn btn-success btn-xs', value='all done')
```

```
form(action='/tasks', method='post')
input(type='hidden', value=locals._csrf, name='_csrf')
div.name
input(type='text', name='name', placeholder='Add a new task')
div.delete
input.btn.btn-primary.btn-sm(type='submit', value='add')
if (tasks.length === 0)
| No tasks.
each task, index in tasks
.item
div.action
form(action='/tasks/#{task._id}', method='post')
input(type='hidden', value=task._id.toString(), name='id')
input(type='hidden', value='true', name='completed')
input(type='hidden', value=locals._csrf, name='_csrf')
input(type='submit', class='btn btn-success btn-xs task-done', value='done')
div.num
span=index+1
|. 
div.name
span.name=task.name
//- no support for DELETE method in forms
//- http://amundsen.com/examples/put-delete-forms/
```

```
//- so do XHR request instead from public/javascripts/main.js
div.delete
a(class='btn btn-danger btn-xs task-delete', data-task-id=task._id.toString(),
data-csrf=locals._csrf) delete
```

Last but not least, comes tasks_completed.jade, which is just a stripped-down version of the tasks.jade file:

```
extends layout
block content
.menu
h2 Menu
ul.nav.nav-pills
li
a(href='/') Home
li
a(href='/tasks') Todo List
li.active
a(href="/tasks/completed") Completed
h1= title
.list
if (tasks.length === 0)
| No tasks.
each task, index in tasks
```

```
.item
div.num
span=index+1
|. 
div.name.completed-task
span.name=task.name
```

Finally, we can customize the look of the app with Less.

Less

As previously mentioned, after applying proper middleware in app.js files, we can put *.less files anywhere under the public folder. Express.js works by accepting a request for some .css file and then attempting to match the corresponding file by name. Therefore, we include *.css files in our jade templates. Here is the content of the todo-express/public/stylesheets/main.less file:

```
* {
font-size:20px;
}
.item {
height: 44px;
width: 100%;
clear: both;
.name {
width: 300px;
}
```

```
.action {
width: 100px;
}
.delete {
width: 100px
}
div {
float:left;
}
}
.home {
margin-top: 40px;
}
.name.completed-task {
text-decoration: line-through;
}
```

To run the application, start MongoDB with $ mongo, and, in a new terminal window, execute $ node app and go to http://localhost:3000/.

In your terminal window, you should see something like this:

```
Express server listening on port 3000
GET / 200 30.448 ms - 1408
GET /stylesheets/style.css 304 7.196 ms - -
```

```
GET /javascripts/jquery.js 304 17.677 ms - -
GET /javascripts/main.js 304 27.151 ms - -
GET /stylesheets/main.css 200 453.584 ms - 226
GET /bootstrap/bootstrap.css 200 458.293 ms - 98336
```

Summary

You've learned how to use MongoDB, Jade, and Less. This Todo app is considered traditional, because it doesn't rely on any front-end framework and it renders HTML on the server. This was done intentionally to show how easy it is to use Express.js for such tasks. In modern-day development, people often leverage some sort of REST API server architecture with a front-end client built with Backbone.js, AngularJS, Ember.js, or something comparable (see http://todomvc.com).