EBook Gratuito

APPRENDIMENTO sails.js

Free unaffiliated eBook created from
Stack Overflow contributors.

#sails.js

Sommario

Di

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: sails-js

It is an unofficial and free sails.js ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official sails.js.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capitolo 1: Iniziare con sails.js

Osservazioni

sails.js è un framework web MVC (Model View Controller) per node.js che emula framework MVC familiari come Ruby on Rails. sails.js è basato su Express e fornisce il supporto websocket tramite socket.io.

sails.js fornisce una serie di convenzioni e configurazioni predefinite per avviare rapidamente un nuovo progetto di sito web. È altamente configurabile e consente di sovrascrivere facilmente le convenzioni di default.

sails.js viene fornito con un ORM chiamato Waterline che astrae l'accesso ai dati. Waterline consente di utilizzare vari datastore come MySQL, PostgreSQL, MongoDB, Redis, ecc. E di avere un'API chiara per lavorare con i dati del modello.

Versioni

Versione	Note di rilascio	changelog	Data di rilascio
0.12.13	Note di rilascio		2017/03/06
0.12.12	Note di rilascio	changelog	2017/03/03
0.12.11	Note di rilascio	changelog	2016/11/24
0.12.10	Note di rilascio	changelog	2016/11/17
0.12.9	Note di rilascio	changelog	2016/11/02
0.12.8	Note di rilascio	changelog	2016/10/22
0.12.7	Note di rilascio	changelog	2016/10/06
0.12.6	Note di rilascio	changelog	2016/09/28
0.12.5	Note di rilascio	changelog	2016/09/28
0.12.4	Note di rilascio	changelog	2016/08/01
0.12.3	Note di rilascio	changelog	2016/04/04
0.12.2	Note di rilascio	changelog	2016/04/02
0.12.1	Note di rilascio	changelog	2016/02/15
0.12.0	Note di rilascio	changelog	2016/02/06

Versione	Note di rilascio	changelog	Data di rilascio
0.11.5	Note di rilascio	changelog	2016/02/05
0.11.4	Note di rilascio	changelog	2016/01/06
0.11.3	Note di rilascio	changelog	2015/11/23
0.11.2	Note di rilascio	changelog	2015/09/23
0.11.0	Note di rilascio	changelog	2015/02/11
0.10.5	Note di rilascio	changelog	2014/08/30
0.10.4	Note di rilascio		2014/08/13
0.10.3	Note di rilascio		2014/08/07
0.10.2	Note di rilascio		2014/08/06
0.10.1	Note di rilascio		2014/08/02

Rilasci prima della `0.10.1` omessi dalla lista. Vedi versioni precedenti

Examples

Installazione

Prerequisiti

- nodejs

Per installare l'ultima release stabile di vele con il comando dello strumento della riga di comando seguente comando:

```
$ sudo npm install sails -g
```

A seconda del tuo sistema operativo potresti non aver bisogno di usare `sudo` .

Creare un nuovo progetto

Una volta installato Sails, digita semplicemente

```
$ sails new <project_name>
```

Questo creerà un progetto Sails scheletro in una nuova cartella chiamata <nome_progetto>.

Puoi anche creare un nuovo progetto in una cartella vuota digitando

```
$ sails new
```

Avvia l'app

Una volta che il tuo progetto è stato creato, puoi avviare l'app digitando

```
$ sails lift
```

Per impostazione predefinita, è possibile accedere all'app nel browser sulla porta 1337. L'URL con la porta è mostrato nel terminale.

Un altro modo per avviare l'app Sails è con il comando `node` :

```
$ node app.js
```

Tuttavia, si perdono alcune funzionalità di sviluppo del comando di `lift` come il ricaricamento automatico dell'app quando le risorse e i file di visualizzazione vengono modificati.

Per lo sviluppo puoi anche usare:

```
$ sails console
```

Ciò consente di eseguire il comando direttamente nella riga di comando. È molto utile per il debug dei modelli.

Ciao mondo

Questo esempio mostra come sviluppare la nostra prima applicazione passo dopo passo, assumendo che tu abbia già installato Sails e che sia stato creato un progetto.

1. Creare un file controller vuoto digitando

```
$ sails generate controller hello
```

2. Trova il nuovo file controller su `api/controllers/HelloControllers.js` e aggiungi il metodo `hello` su di esso.

```
module.exports = {

  hello : function (req, res) {
    var myName = 'Luis';
    return res.view('hello' , {name : myName});
    }
}
```

3. Crea un nuovo file di visualizzazione sotto le `views` della cartella denominate `hello.ejs` con il

seguente codice HTML:

```
<html>
    <head></head>
    <body>
        <p>Hello {{}}.</p>
    </body>
</html>
```

4. Definisci una rotta in `config/routes.js` che chiama il metodo `hello` nel controller `HelloController` .

```
'GET /' : 'HelloController.hello',
```

Ora abbiamo implementato tutto il codice necessario per questo esempio. Proviamolo:

1. Avvia il server

```
$ sails lift
```

2. Aprire il browser e digitare `http://localhost:1337` . Se non è in arrivo, controlla l'URL nell'output delle `sails lift` . La porta potrebbe essere diversa.
3. Dovresti vedere il seguente output:

   Ciao Luis

Generazione di progetti di vele senza frontend

Se non c'è bisogno di frontend nel tuo prossimo progetto, puoi eseguire vele nuove con flag aggiuntivo --no-frontend.

```
sails new NameOfProject --no-frontend
```

Questo genererà tutto il necessario per il back-end e tralascerà la vista, le risorse e i file grunt.

Ulteriori informazioni sulla riga di comando e sulle vele: nuovo: http://sailsjs.org/documentation/reference/command-line-interface/sails-new

Leggi Iniziare con sails.js online: https://riptutorial.com/it/sails-js/topic/1205/iniziare-con-sails-js

Capitolo 2: Adattatore di database PostgreSQL per vele

Examples

Installare

È possibile installare l'adattatore PostgreSQL tramite NPM.

```
npm install sails-postgresql
```

Configurazione

È possibile configurare le impostazioni del database in `config/connections.js` .

Ecco un esempio:

```
postgresql: {
  database: 'databaseName',
  host: 'localhost',
  user: 'root',
  password: '',
  port: 5432,
  poolSize: 10,
  ssl: false
};
```

In alternativa, puoi fornire le informazioni di connessione in formato URL:

```
postgresql: {
  url: 'postgres://username:password@hostname:port/database',
  ssl: false
};
```

Leggi Adattatore di database PostgreSQL per vele online: https://riptutorial.com/it/sails-js/topic/4676/adattatore-di-database-postgresql-per-vele

Capitolo 3: Adattatore MongoDB per vele

Examples

Configurazione

È possibile configurare le impostazioni del database in `config/connections.js` .

Esempio:

```
someMongoDb: {
  adapter: 'sails-mongo',
  host: 'localhost', // defaults to `localhost` if omitted
  port: 27017, // defaults to 27017 if omitted
  user: 'username_here', // or omit if not relevant
  password: 'password_here', // or omit if not relevant
  database: 'database_name_here' // or omit if not relevant
}
```

In alternativa, puoi specificare la tua configurazione di Mongo come URL

```
someMongoDb: {
  adapter: 'sails-mongo',
  url: mongodb://username:password@hostname:port/database
}
```

Installazione

Installa da NPM.

```
npm install sails-mongo --save
```

Leggi Adattatore MongoDB per vele online: https://riptutorial.com/it/sails-js/topic/7047/adattatore-mongodb-per-vele

Capitolo 4: API Blueprint

Osservazioni

Come funziona l'API di Blueprint?

Quando le vele iniziano a usare le `sails lift` , le vele guardano per vedere se hai un controller definito. Nel nostro esempio, abbiamo un controller, il controller utente. Sails fornisce quindi l'accesso alle azioni blueprint per questo controller utente come se fossero state create direttamente nel controller. Anche le vele creano automaticamente percorsi di progetto al momento del sollevamento del server. Quindi, anche se in `/config/routes.js` non sono definiti `/api/controllers/UserController.js` , e in `/api/controllers/UserController.js` non viene definita alcuna azione esplicita, dopo aver `/api/controllers/UserController.js` il server tutte queste rotte e azioni sono disponibili per l'uso.

Examples

Percorsi Blueprint

Quando si esegue il `sails lift` con i modelli abilitati, il framework controlla i controller, i modelli e la configurazione per collegare automaticamente determinati percorsi. Questi percorsi impliciti del modello consentono alla tua app di rispondere a determinate richieste senza che tu debba vincolare manualmente tali percorsi nel tuo file `config/routes.js` . Per impostazione predefinita, i percorsi del modello puntano alle corrispondenti *azioni del* progetto, ognuna delle quali può essere sostituita con un codice personalizzato.

Ci sono tre tipi di rotte blueprint in Sails:

- **Percorsi RESTful** , dove il percorso è sempre `/:model/:id?` . Quando il modello `User` e il controller sono definiti, il blueprint associa implicitamente le rotte RESTful nel seguente modo:

```
'GET /user/:id?': {
  controller: 'User',
  action: 'find'
},
'POST /user': {
  controller: 'User',
  action: 'create'
},
'PUT /user/:id?': {
  controller: 'User',
  action: 'update'
},
'DELETE /user/:id?': {
  controller: 'User',
  action: 'destroy'
}
```

Queste rotte utilizzano il verbo HTTP per determinare l'azione da eseguire anche se il percorso è lo stesso. Quindi, una richiesta `POST` a `/user` creerà un nuovo utente, una richiesta `PUT` a `/user/123` aggiornerà l'utente con la chiave primaria 123 e una richiesta `DELETE` a `/user/123` cancellerà l'utente la cui chiave primaria è 123. In un ambiente di produzione, le rotte RESTful dovrebbero generalmente essere protette da policy per evitare accessi non autorizzati.

- **Percorsi di scelta rapida** , in cui l'azione da intraprendere è codificata nel percorso. Per il nostro modello di utente e controller, Sails associa implicitamente quattro percorsi di scelta rapida.

```
'GET /user/find/:id?': {
  controller: 'User',
  action: 'find'
},
'GET /user/create/:id?': {
  controller: 'User',
  action: 'create'
},
'GET /user/update/:id?': {
  controller: 'User',
  action: 'update'
},
'GET /user/destroy/:id?': {
  controller: 'User',
  action: 'destroy'
}
```

Ad esempio, il collegamento `/user/create?name=joe` crea un nuovo utente, mentre `/user/update/1?name=mike` aggiorna il campo del nome dell'utente # 1. Nota che questi percorsi rispondono solo alle richieste `GET` . I percorsi di scelta rapida sono molto utili per lo sviluppo, ma in genere dovrebbero essere disabilitati in un ambiente di produzione. Non è progettato per essere utilizzato in produzione.

- **Percorsi di azione** , che creano automaticamente percorsi per le azioni del controller personalizzato. Ad esempio, lascia che la `query` sia un'azione personalizzata definita nel controller utente. Quindi seguire le rotte sarebbe implicitamente disponibile per le vele -

```
'GET /user/query/:id?': {
  controller: 'User',
  action: 'query'
},
'POST /user/query/:id?': {
  controller: 'User',
  action: 'query'
},
'PUT /user/query/:id?': {
  controller: 'User',
  action: 'query'
},
'DELETE /user/query/:id?': {
  controller: 'User',
  action: 'query'
}
```

Se la richiesta è fatta in `/user/query/:id?` percorso quindi indipendente sul verbo HTTP l'azione sarebbe la stessa. A differenza dei percorsi RESTful e di scelta rapida, i percorsi di azione *non* richiedono che un controller abbia un file modello corrispondente. Il che significa che se si definisce un controller nel file `/api/controllers/FooController.js` ma nessun modello nel file `/api/models/Foo.js` , non ci sarebbe nessun RESTful o percorso di scorciatoia con `/foo` ma ci saranno comunque percorsi di azione disponibile per l'uso.

Ordine di percorsi corrispondenti

Quando arriva una richiesta, le vele corrisponderanno prima alla rotta contro rotte esplicitamente definite. Se corrisponde, non viene eseguita alcuna ulteriore corrispondenza e viene eseguita l'azione corrispondente. Ma se non corrisponde, la rotta viene confrontata in primo luogo con le rotte di azione del modello, se non corrisponde a quelle di riposo e se non corrisponde a quelle di collegamento. Quindi se il tuo file `/config/routes.js` ha qualche voce come la seguente-

```
'/user/query/:id?': {
  controller: 'User',
  action: 'find'
}
```

Quindi non è possibile aspettarsi che i percorsi di azione della `query` funzionino. Poiché la stessa route del percorso di azione della `query` dovrebbe essere confrontata con le rotte di definizione esplicita e l'azione di `find` del controller utente verrà eseguita.

Azioni blueprint

Le azioni Blueprint (da non confondere con i *percorsi di* azione blueprint) sono azioni generiche progettate per funzionare con qualsiasi controller che ha un modello con lo stesso nome (ad esempio `ParrotController` avrebbe bisogno di un modello `Parrot`). Pensa a loro come il comportamento predefinito per la tua applicazione. Ad esempio, se hai un modello `User.js` e un controller `UserController.js` vuoto, `find` , `create` , `update` , `destroy` , `populate` , `add` e `remove` azioni implicitamente, senza doverle scrivere.

Per impostazione predefinita, i percorsi RESTful e le rotte di scelta rapida di blueprint sono associati alle rispettive azioni blueprint corrispondenti. Tuttavia, qualsiasi azione di modello può essere sovrascritta per un particolare controller creando un'azione personalizzata in quel file di controller (ad es. `ParrotController.find`). In alternativa, puoi sovrascrivere l'azione del modello *ovunque nella tua app* creando la tua azione personalizzata.

Sails viene fornito con le seguenti azioni blueprint:

- trova
- trova uno
- creare
- aggiornare
- distruggere
- popolare

- Inserisci
- rimuovere

Disabilitare rotte Blueprint

- **Base globale** : la configurazione dell'API Blueprint è definita nel file `/config/blueprint.js` . Da qui puoi abilitare o disabilitare tutti e tre i tipi di rotte blueprint per tutti i controller. Ad esempio, se si desidera disabilitare i percorsi di scelta rapida del modello per tutti i controller, ma si desidera mantenere attive sia le rotte di azione che quelle di riposo, allora il tuo `/config/blueprint.js` dovrebbe essere come questo -

```
module.exports.blueprints = {
  action: true,
  rest: true,
  shortcut: false
}
```

- **In base al controller** : è inoltre possibile sovrascrivere qualsiasi impostazione da `/config/blueprints.js` in base al controller definendo una chiave "_config" nella definizione del controller e assegnandole un oggetto di configurazione con le sostituzioni per le impostazioni in questo file. Ad esempio, se si desidera avere percorsi di scelta rapida abilitati solo per il controller utente ma non per altri controller, con la configurazione del modello di cui sopra è necessario avere la seguente coppia di valori chiave nel controller utente.

```
module.exports = {
  _config: {
    actions: true,
    shortcuts: true,
    rest: true
  }
}
```

Leggi API Blueprint online: https://riptutorial.com/it/sails-js/topic/3749/api-blueprint

Capitolo 5: Autenticazione del token Web JSON con Sails

Examples

Configurazione

Primo passo

Dobbiamo creare un servizio chiamato *jwToken* . Vai alla directory `api/services` e crea `jwToken.js` .

```
'use strict';

const jwt = require('jsonwebtoken'),
      tokenSecret = "secretissecret";

module.exports = {
  // Generates a token from supplied payload
  issue(payload) {
    return jwt.sign(
      payload,
      tokenSecret, // Token Secret that we sign it with
      {
        expiresIn: "30 days" // Token Expire time
      });
  },

  // Verifies token on a request
  verify(token, callback) {
    return jwt.verify(
      token, // The token to be verified
      tokenSecret, // Same token we used to sign
      {}, // No Option, for more see https://github.com/auth0/node-
jsonwebtoken#jwtverifytoken-secretorpublickey-options-callback
      callback //Pass errors or decoded token to callback
    );
  }
};
```

Passo due

Cripta la nostra password usando `bcrypt` . Vai a `api/models/User.js`

```
'use strict';
const bcrypt = require('bcrypt');

module.exports = {

    attributes: {
```

```
        // your code...
    },

    // Here we encrypt password before creating a User
    beforeCreate(values, next) {
        bcrypt.genSalt(10, (err, salt) => {
            if (err) {
                sails.log.error(err);
                return next();
            }

            bcrypt.hash(values.password, salt, (err, hash) => {
                if (err) {
                    sails.log.error(err);
                    return next();
                }
                values.encryptedPassword = hash; // Here is our encrypted password
                return next();
            });
        });
    },

    comparePassword(password, encryptedPassword) {

        return new Promise(function(resolve, reject) {
            bcrypt.compare(password, encryptedPassword, (err, match) => {
                if (err) {
                    sails.log.error(err);
                    return reject("Something went wrong!");
                }
                if (match) return resolve();
                else return reject("Mismatch passwords");
            });
        });
    }
};
```

Fase tre

Crea criterio di *autorizzazione* per verificare se un utente ha un token valido nell'intestazione della richiesta. Vai a `api/policies` e crea `isAuthorized.js` .

```
'use strict';

module.exports = (req, res, next) => {
  let token;


  if (req.headers && req.headers.token) {
    token = req.headers.token;
    if (token.length <= 0) return res.json(401, {err: 'Format is Authorization: Bearer
[token]'});

  } else if (req.param('token')) {
    token = req.param('token');
    // We delete the token from param to not mess with blueprints
    delete req.query.token;
```

```
  } else {
    return res.json(401, {err: 'No Authorization header was found'});
  }

  jwToken.verify(token, function (err, token) {
    if (err) return res.json(401, {err: 'Invalid Token!'});
    req.token = token; // This is the decrypted token or the payload you provided
    next();
  });
};
```

Fase quattro

Utilizziamo config / policies.js per proteggere i nostri controllori

```
module.exports.policies = {

  '*': ['isAuthorized'], // Everything resctricted here
  'UserController': { // Name of your controller
    'create': true // We dont need authorization here, allowing public access
  }
};
```

Passo cinque

Mettiamo alla prova la nostra implementazione. Vai a `api/controllers` e crea `UserController.js`

```
'use strict';

module.exports = {
    create(req, res) {
        const data = req.body;
        if (data.password !== data.confirmPassword) return res.badRequest("Password not the
same");

        User.create({
                email: data.email,
                password: data.password,
                name: data.name
                    //etc...
            })
            .then((user) => {
                res.send({ token: jwToken.issue({ id: user.id }) }); // payload is { id:
user.id}
            })
            .catch((err) => {
                sails.log.error(err);
                return res.serverError("Something went wrong");
            });
    },

    login(req, res) {
        const data = req.body;
```

```
        if (!data.email || !data.password) return res.badRequest('Email and password
required');

        User.findOne({ email: email })
            .then((user) => {
                if (!user) return res.notFound();

                User.comparePassword(password, user.password)
                    .then(() => {
                        return res.send({ token: jwToken.issue({ id: user.id }) })
                    })
                    .catch((err) => {
                        return res.forbidden();
                    });
            })
            .catch((err) => {
                sails.log.error(err);
                return res.serverError();
            });
    }
};
```

Installazione

Abbiamo bisogno di due dipendenze:

1. **bcrypt** per la crittografia `npm install bcrypt --save`
2. **Token Web JSON** `npm install jsonwebtoken --save`

Leggi Autenticazione del token Web JSON con Sails online: https://riptutorial.com/it/sails-js/topic/7050/autenticazione-del-token-web-json-con-sails

Capitolo 6: Configurazione di mysql con sails.js

Examples

Come configurare la connessione al database mysql in sails.js

Per fare ciò, innanzitutto individuare la cartella di configurazione nella root. Quindi apri `connections.js`

Individuare

```
// someMysqlServer: {
//   adapter: 'sails-mysql',
//   host: 'YOUR_MYSQL_SERVER_HOSTNAME_OR_IP_ADDRESS',
//   user: 'YOUR_MYSQL_USER', //optional
//   password: 'YOUR_MYSQL_PASSWORD', //optional
//   database: 'YOUR_MYSQL_DB' //optional
// },
```

Rimuovi il commento da queste righe.

Dare un nome adatto per il connettore come questo unMysqlServer a mysql_connection o qualsiasi altro nome come si desidera

```
mysql_connection: {
  adapter: 'sails-mysql',
  host: '127.0.0.1', // can user localhost or mysql connection either
  user: 'root', // your mysql username
  password: 'xxxxxxxxx', // your mysql password
  database: 'your database name here' // database name
},
```

Salvare il file

Vai alla tua cartella principale ed esegui il seguente comando:

```
$ npm install sails-mysql --save
```

Nota: eseguendo il comando sopra stiamo installando il pacchetto driver mysql per sails.js.

Il tuo è fatto.

Ora puoi usare mysql_connection come nome di connessione nella configurazione del tuo modello. Quando sollevi la tua app utilizzando il seguente comando:

```
$ sails lift
```

lo schema del modello verrà automaticamente aggiornato nel database MySQL

Leggi Configurazione di mysql con sails.js online: https://riptutorial.com/it/sails-js/topic/5317/configurazione-di-mysql-con-sails-js

Capitolo 7: Controller

Osservazioni

I controller (la **C** in **MVC**) sono gli oggetti principali dell'applicazione Sails che sono responsabili di rispondere alle richieste da un browser Web, un'applicazione mobile o qualsiasi altro sistema in grado di comunicare con un server. Spesso fungono da intermediari tra i tuoi modelli e le tue visualizzazioni. Per molte applicazioni, i controller contengono la maggior parte della logica di business del progetto.

Examples

Sintassi ES2015

```
'use strict';

// This is an example of a /api/controllers/HomeController.js
module.exports = {
  // This is the index action and the route is mapped via /config/routes.js
  index(req, res) {
    // Return a view without view model data
    // This typically will return the view defined at /views/home/index.<view engine
extension>
    return res.view();
  },
  foo(req, res) {
    // Return the 'foo' view with a view model that has a `bar` variable set to the query
string variable `foobar`
    return res.view({
      bar: req.param('foobar'),
    });
  },
};
```

Utilizzo dei generatori ES2015 con co.js

```
'use strict';

const co = require('co');

module.exports = {
  // This is the index action and the route is mapped via /config/routes.js
  index(req, res) {
    co(function* index() {
      // Return a view without view model data
      // This typically will return the view defined at /views/home/index.<view engine
extension>
      return res.view();
    }).catch(res.negotiate); // Catch any thrown errors and pass the error to the `negotiate`
policy.
  },
  foo(req, res) {
```

```
    co(function* foo() {
      // Get an array of `FooBar` items from the database
      const items = yield FooBar.find();

      // Return the 'foo' view with a view model containing the array of `FooBar` items
      return res.view({
        items,
      });
    }).catch(res.negotiate); // Catch any thrown errors and pass the error to the `negotiate`
policy.
  },
};
```

Leggi Controller online: https://riptutorial.com/it/sails-js/topic/3521/controller

Capitolo 8: Modelli

Osservazioni

Sails viene installato con un potente ORM / ODM chiamato Waterline, uno strumento agostino-datastore che semplifica notevolmente l'interazione con uno o più database. Fornisce un livello di astrazione sopra il database sottostante, consentendo di interrogare e manipolare facilmente i dati senza scrivere codice di integrazione specifico del fornitore.

Examples

Modello base

Questo esempio mostra come definire un modello semplice in Sails.js

È possibile generare un file di modello vuoto digitando

```
sails generate model car
```

Troverai il nuovo file `Car.js` in `api/models/` .

Quindi, inserisci alcuni dettagli.

```
modules.exports = {

  tableName : 'cars',
  connection : 'mongodb',

  attributes : {

    id : {
      type : 'integer',
      unique : true,
      primaryKey : true,
      autoIncrement : true
    },

    brand : {
      type : 'string',
      size : 25
    },

    description : {
      type: 'text',
      defaultsTo : ''
    },

    price : {
      type : 'float',
      required : true
    },
```

```
    seats : {
      type : 'integer'
    },

    sell_date : {
      type : 'datetime'
    },

    has_cooler : {
      type : 'boolean',
      columnName : 'cooler'
    },

    chassis_number : {
      unique : true,
      type : 'string'
    },

    color : {
      type : 'string',
      enum: ['white', 'red', 'black']
    }

  }

};
```

L'esempio sopra riportato utilizza quasi tutte le possibili opzioni di modello, che sono illustrate di seguito.

1. tableName

Questo parametro definisce il nome della tabella che verrà creata nel database. Se non definito, verrà utilizzato il nome del modello (`car` in questo esempio).

2. connessione

Questo particolare definisce la connessione al database utilizzata per il modello. I dettagli di tale connessione sono definiti sotto la chiave `mongodb` all'interno di `config/connections.js` . Ecco il formato di una connessione:

```
mongodb : {

  // The driver that connect our models with the database
  adapter : '<adapter>',

  // The database parameters
  user : '<username>',
  port : <port>,
  host : '<host>',
  database : '<database>'

}
```

3. attributi

Ogni attributo fa riferimento a una colonna nella tabella del database per il modello. In questo esempio, verranno create nove colonne. Ogni colonna può essere configurata con uno o più dei seguenti tasti:

- **tipo** : il tipo di dati della colonna. Questa pagina elenca tutti i tipi disponibili.
- **unique** : Se true, si verificherà un errore se si tenta di creare un oggetto che abbia lo stesso valore per questa colonna come già nel database.
- **primaryKey** : se true, la colonna funzionerà come chiave primaria.
- **autoIncrement** : una sequenza verrà associata alla colonna con un numero incrementale automatico che inizia da 0.
- **dimensione** : la lunghezza massima della colonna.
- **richiesto** : se true, non può essere nullo.
- **columnName** : configura la colonna nel database, che **assume come** valore predefinito il nome dell'attributo.
- **enum** : possiamo impostare una serie di opzioni possibili per un attributo.

Leggi Modelli online: https://riptutorial.com/it/sails-js/topic/4646/modelli

Capitolo 9: Routing

Osservazioni

I percorsi sono regole che indicano a Sails cosa fare di fronte a una richiesta in arrivo.

Le rotte sono definite in `config/routes.js` . L'ordine dei percorsi è significativo, poiché i percorsi sono abbinati dall'alto verso il basso. Ciò significa che se si dispone di una rotta specifica che potrebbe anche essere abbinata a un percorso con caratteri jolly, il percorso specifico deve essere definito sopra il percorso con caratteri jolly.

Quando una richiesta entra nella tua applicazione, sails.js acquisisce tutti i parametri forniti e li rende disponibili come `params` sull'oggetto richiesta.

Le proprietà nell'oggetto di destinazione del percorso verranno passate al gestore di route nell'oggetto req.options. Le seguenti sono proprietà riservate che possono influire sul comportamento del gestore di instradamento:

Proprietà	Tipi di target applicabili	Tipo di dati	Dettagli
skipAssets	tutti	booleano	Impostato su `true` se *non si* desidera che la route corrisponda agli URL con punti all'interno (es. MyImage.jpg). Ciò manterrà le rotte con la notazione jolly dagli URL corrispondenti delle risorse statiche. Utile durante la creazione di slug URL.
skipRegex	tutti	regexp	Se saltare tutti gli URL contenenti un punto è troppo permissivo, o è necessario saltare il gestore di una rotta in base a diversi criteri, è possibile utilizzare `skipRegex` . Questa opzione consente di specificare un'espressione regolare o una matrice di espressioni regolari per abbinare l'URL della richiesta a; se una delle corrispondenze ha esito positivo, il gestore viene saltato. Si noti che, a differenza della sintassi per il binding di un gestore con un'espressione regolare, `skipRegex` aspetta _errual oggetti RegExp, non stringhe.
gente del posto	controller, vista, blueprint,	Dizionario	Imposta le variabili locali predefinite da passare a qualsiasi vista resa durante la gestione della richiesta.

Proprietà	Tipi di target applicabili	Tipo di dati	Dettagli
	risposta		
cors	tutti	Dizionario o booleano o stringa	Specifica come gestire le richieste per questa rotta da un'origine diversa.
popolare	planimetria	booleano	Indica se i risultati di un'azione "trova" o "findOne" devono avere i campi del modello associati compilati. Il valore predefinito è impostato in `config/blueprints.js` .
saltare , **limitare** , **ordinare** , **dove**	planimetria	Dizionario	Imposta i criteri per il progetto "trova".

Examples

Definizioni di route RESTful personalizzate

```
module.exports.routes = {
    'GET /foo': 'FooController.index',
    'GET /foo/new': 'FooController.new',
    'POST /foo/create': 'FooController.create',
    'GET /foo/:id/edit': 'FooController.edit',
    'PUT /foo/:id/update': 'FooController.update',
    'GET /foo/:id': 'FooController.show',
    'DELETE /foo/:id': 'FooController.delete',
};
```

Reindirizzare

```
module.exports.routes = {
  '/foo': '/bar',
  'GET /google': 'http://www.google.com'
};
```

Definisci variabile personalizzata per tutte le viste

```
module.exports.routes = {
  // This function will be executed for all http verbs on all urls
  'all /*', function (req, res, next) {
    // Expose the function `fooBar` to all views (via the locals object)
    res.locals.fooBar = function (arg1) {
      return 'foobar' + arg1;
    };
```

```
  },
};
```

Salta le risorse (URL con punti in esse) dal percorso con caratteri jolly

```
module.exports.routes = {
  'GET /foo/*': {
     fn: function(req, res) {
       res.send("FOO!");
     },
     skipAssets: true
  },
};
```

Percorsi con RegEx

```
module.exports.routes = {
    // sends matching regex (few patterns) as 'page' param
    'r|/foo/([0-9]+)|page': 'FooController.get',
    'r|/foo/(.*)|page': 'FooController.get',
    'r|/foo/(\\w+)|page': 'FooController.get'
};
```

Lumache di URL

```
module.exports.routes = {
  'GET /blog/:year/:month/:day/:posttitle/': 'BlogController.showPost',
  'GET /blog/:year/:month/:day/': 'BlogController.showDayArchive',
  'GET /blog/:year/:month/': 'BlogController.showMonthArchive',
  'GET /blog/:year/': 'BlogController.showYearArchive',
};
```

È quindi possibile accedere ai parametri trasmessi nell'URL nelle corrispondenti azioni del controller utilizzando `req.param('year')` , `req.param('month')` ecc.

Ad esempio, una richiesta `GET` a `/blog/2016/08/` `BlogController.showMonthArchive` `/blog/2016/08/` attiva l'azione del controller `BlogController.showMonthArchive` , con `req.param('year')` con valore `2016` e `req.param('month')` con valore `08` .

Leggi Routing online: https://riptutorial.com/it/sails-js/topic/4577/routing

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con sails.js	Bonanza, Community, hlozancic, Jim Geurts, Lost in OWL, Luis González, mouche, Sebastialonso, Viktor, Yann Bertrand
2	Adattatore di database PostgreSQL per vele	Luis González, mouche
3	Adattatore MongoDB per vele	SkyQ
4	API Blueprint	taufique
5	Autenticazione del token Web JSON con Sails	SkyQ, Viktor
6	Configurazione di mysql con sails.js	vijeeshin, Yann Bertrand
7	Controller	Jim Geurts
8	Modelli	Luis González, mouche, Viktor
9	Routing	Bonanza, Jim Geurts, Viktor