linzjs Documentation

Release query-plugin

Jan 23, 2018

Developer documentation

Linz is a framework for creating administration interfaces. Linz is not a CMS, but is capable of CMS like functionality. Linz is a good choice of framework when the administration interface *is* the website itself.

Linz is built on Node.js, Express and MongoDB.

Linz is quite new and under rapid development. It is used quite successfully in a number of production sites however. This documentation is the first effort in making this open source project accessible to more developers.

CHAPTER 1

Getting started with Linz

This will help you create a new Linz-based website. If you'd like to develop Linz itself, see *Getting started with Linz development*.

While we're working on our documentation, you can get started with Linz via our example project, see *mini-twitter*.

Linz tries to force as little new syntax on you as possible. Of course, this is unavoidable in certain situations, and there are some conventions you'll need to learn. We've tried to keep them as simple as possible.

Linz does make use of many other open source tools, such as Mongoose or Express. Linz tries to keep the usage of these tools as plain and simple as possible. Linz doesn't wrap, customise or prettify syntax for other libraries/tools used within Linz. The three primary opensource tools that Linz relies on are:

- Express
- Mongoose
- Passport

The following will be a general overview of some of the core concepts of Linz.

1.1 Singleton

When you require Linz, you're returned a singleton. This has the advantage that no matter where you require Linz, you get the same Linz instance.

1.2 Initialization

Linz must be initialized. During initialization, Linz accepts an options object with any of the following optional keys:

- `express`: An initialized Express intance.
- `passport`: An initialized Passport instance.
- `mongoose`: An initialized Mongoose instance.

- `options`: An options object to customise Linz.

For example:

```
var express = require('express'),
    mongoose = require('mongoose'),
    passport = require('passport'),
    linz = require('linz');

linz.init({
    express: express(),
    mongoose: mongoose
    passport: passport,
    options: {
        'load configs': false
    }
});
```

If neither an initialized instance of Express, Passport or Mongoose, nor an options object have been passed, Linz will create them for you:

```
// Use anything that has been passed through, or default it as required.
this.app = opts.express || express();
this.mongoose = opts.mongoose || require('mongoose');
this.passport = opts.passport || require('passport');

// overlay runtime options, these will override linz defaults
this.options(opts.options || {});
```

1.2.1 Options object

An object can be used to customize Linz. For example:

```
linz.init({
    options: {
        'mongo': `mongodb://${process.env.MONGO_HOST}/db`
    }
});
```

You can read more about *Linz defaults*.

1.3 Events

The Linz object is an event emitter, and will emit the `initialized` event when Linz has finished initializing.

It will also emit an event whenever a configuration is set (i.e. using the *linz.set* method). The name of the event will be the same as the name of the configuration that is set.

A common pattern for setting up Linz, using the event emitter, is as follows:

server.js:

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

linz.on('initialised', require('./app'));
```

```
// Initialize Linz.
linz.init({
    options: {
        mongo: `mongodb://${process.env.DB_HOST || 'localhost'}/lmt`,
        'user model': 'mtUser'
    }
});
```

app.js:

```
var http = require('http'),
  linz = require('linz'),
  routes = require('./routes'),
  port = process.env.APP_PORT || 4000;

module.exports = function () {

  // Mount routes on Express.
  linz.app.get('/', routes.home);
  linz.app.get('/bootstrap-users', routes.users);

  // Linz error handling midleware.
  linz.app.use(linz.middleware.error);

  // Start the app.
  http.createServer(linz.app).listen(port, function(){
    console.log('');
    console.log(`mini-twitter app started and running on port ${port}`);
  });

};
```

1.4 Directory structure

Linz expects a common directory structure. If provided, it will load content from these directories. These directories should live alongside your Node.js entry point file (i.e. `node server.js`).

- `models`: a directory of model files.
- `schemas`: a directory of schemas, which are used as nested schemas within a model.
- `configs`: a directory of config files.

You can read more about each of the above and what Linz expects in the documentation covering each area.

CHAPTER 2

API

Linz contains many useful APIs to simplify development.

At present, we're working on a long term effort to expose all of Linz's functionality via APIs and have Linz use these APIs itself. This will provide the most flexibility to customise Linz, without having to do the heavy lifting yourself when choosing to change something about Linz.

You can access the Linz APIs via `linz.api`.

2.1 Views

The methods exposed via the `linz.api.views` namespace have functionality centered around working with Linz's views.

You can use these methods to render your own content, within a Linz template.

2.1.1 views.getScripts(req, res, scripts = [])

Get the scripts that Linz uses for a particular route.

2.1.2 views.getStyles(req, res, scripts = [])

Get the styles that Linz uses for a particular route.

2.1.3 views.notification(noty)

Takes an object, and applies noty defaults to it, making it easy to create noty objects for notifications. You can ready more about Noty options.

2.1.4 views.render(options, callback)

Render some HTML, within a Linz template. Useful for developing completely custom page content, without having to provide the Linz basics such as navigation, log out controls, etc.

Parameters:

options: object

`options` object is used to pass the HTML that should be rendered within the Linz template. The following table describes the properties of the `options` object.

Property	Type	Description
`header`	HTML	This is the header of the page. You'll need to include wrapping div elements in your HTML to make it look like the default.
`body`	HTML	This is the body of the page. You'll need to include wrapping div elements in your HTML to make it look like the default.
`page`	HTML	If provided, header and body are ignored, allowing completely custom content.
`script`	HTML	This is intended to be `<script>` tags placed at the bottom (just above `</body>`).
`template`	String	Either `wrapper` or `wrapper-preauth`. This is the template to use to wrap the provided HTML. Defaults to `layout`.

callback

`callback` can be one of two things:

- A standard callback `function` in the format `callback(err, html)`.
- An Express response (`res`) object.

If an Express reponse object is provided, Linz will automatically call `res.render` with the rendered HTML.

Example

```
const linz = require('linz');

module.exports = function (req, res, next) {

  const locals = {
    header: '<div class="col-xs-12"><div class="model-title"><h1>Header.</h1></div></
↪div>',
    body: '<div class="container linz-container index"><div class="col-xs-12"><p>Body.
↪</p></div></div>'
  };

  linz.api.views.render(locals, req, res, (err, html) => {

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

    return res.send(html);
```

```
  });

};
```

2.1.5 views.viewPath(view)

Returns the complete path to one of Linz's views.

Parameters:

view: string

`view` is a string that is the name of a view found within Linz's views directory.

CHAPTER 3

Linz defaults

Linz has a bunch of defaults, which can be use to alter Linz and the way it works. The coverage of the defaults and what they actually do, vary from default to default.

This document outlines the defaults and how they can be used to alter Linz.

Please note: this document is a work in progress and not all defaults have been documented. For a complete list of customizations you can make, view Linz's defaults.

3.1 Setting defaults

There are two ways to alter the defaults:

- At init time.
- Any other time using the `linz.set` function.

Read about the *Options object* for more information about how to set the defaults at init time.

Alternatively, you can set a default using the `linz.set` function:

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

linz.on('initialised', require('./app'));

linz.set('mongo', `mongodb://${process.env.MONGO_HOST}/db`);
```

3.2 The defaults

The following lists the defaults that you can use to customise Linz.

3.2.1 customAttributes

The `customAttributes` default allows you to customise the HTML attributes applied to the `body`. This can be useful for a range of things including targeting styles specific to a custom attribute, or supplying information about the user which can be used by JavaScript widgets.

To use `customAttributes` define a function with the following signature:

```
/**
 * @param {Object} req  A HTTP request object.
 * @return Array
 */
customAttributes (req)
```

The function should return an array of objects containing attributes in a name/value pair:

```
customAttributes (req) => {

  const attributes = [];

  if (req.user && req.user.group) {
    attributes.push({
      name: 'data-linz-usergroup',
      value: req.user.group
    });
  }

  return attributes;

}
```

This will result in a `body` tag with custom attributes on all Linz pages:

```
<body data-linz-usergroup='20'>
```

3.2.2 scripts

The `scripts` default allows you to customise the external JavaScripts that are loaded on each page in Linz.

To use `scripts` define a function with the following signature:

```
/**
 * @param {Object} req A HTTP request object.
 * @param {Object} res A HTTP response object.
 * @return {Promise} Resolves with an array of script objects.
 */
scripts (req, res)
```

The function should return an array of objects containing the same HTML attributes as the `<script>` tag:

```
scripts (req, res) => {

  return Promise.resolve(res.locals.scripts.concat([
    {
      crossorigin: 'anonymous',
      integrity: 'sha256-5YmaxAwMjIpMrVlK84Y/+NjCpKnFYa8bWWBbUHSBGfU=',
      src: '//cdnjs.cloudflare.com/ajax/libs/bootstrap-datetimepicker/4.17.47/js/
↪bootstrap-datetimepicker.min.js',
```

```
    },
  ]));

}
```

`res.locals.scripts` contains all the scripts used by Linz, be careful when removing/updating these as it could break functionality within Linz. You should use the existing array as the array that is resolved with the promise because it will replace `res.locals.scripts`, not append to it.

The script objects can contain an additional `inHead` boolean option to optionally load the script in the head tag.

You can also supply a `content` property, which if provided, will add the value of the `content` property within the script open and close tags.

3.2.3 styles

The `styles` default allows you to customise the external CSS stylesheets that are loaded on each page in Linz.

To use `styles` define a function with the following signature:

```
/**
 * @param {Object} req A HTTP request object.
 * @param {Object} res A HTTP response object.
 * @return {Promise} Resolves with an array of style objects.
 */
styles (req, res)
```

The function should return an array of objects containing the same HTML attributes as the `<link>` tag:

```
styles (req, res) => {

  return Promise.resolve(res.locals.styles.concat([
    {
      crossorigin: 'anonymous',
      href: '//cdnjs.cloudflare.com/ajax/libs/bootstrap-datetimepicker/4.17.47/css/
↪bootstrap-datetimepicker.min.css',
      integrity: 'sha256-yMjaV542P+q1RnH6XByCPDfUFhmOafWbeLPmqKh11zo=',
      rel: 'stylesheet',
    },
  ]));

}
```

`res.locals.styles` contains all the styles used by Linz, be careful when removing/updating these as it could break functionality within Linz. You should use the existing array as the array that is resolved with the promise because it will replace `res.locals.styles`, not append to it.

You can also supply a `content` property, which if provided, will add the value of the `content` property within a `style` open and close tags.

3.2.4 mongoOptions

Mongoose's default connection logic is deprecated as of 4.11.0. `mongoOptions` contains the minimum default connection logic required for a connection:

'mongoOptions': { useMongoClient: true

```
    }
```

See Mongoose connections. for more details and configurations.

3.2.5 404

The *404* default allows you to pass in your own 404 html.

To use `404` define a function with the following signature:

```
/**
 * @param {Object} req A HTTP request object.
 * @return {Promise} Resolves with the html.
 */
404 (req) => Promise.resolve(html)
```

The function should return a Promise that resolves with the html string.

CHAPTER 4

Models

One of the primary reasons to use Linz is to ease model development and scaffold highly customizable interfaces for managing these models. Linz provides a simple DSL you can use to describe your model. Using the content of your DSL, Linz will scaffold an index, overview, edit handlers and provide a basic CRUD HTTP API for your model.

All Linz models are bootstrapped with two properties (created by Linz):

- `dateCreated` with a label of *Date created*.
- `dateModified` with a label of *Date modified*.

Note: Linz will display your models in a list. The label used for each record is derived from the title field, or a virtual title field if one does not exist in your schema.

If your model has a title field, you don't have to do anything. If your model doesn't have a title field, you can tell Linz about another field in the schema that you would like used to derive a value and label for each record. The title is the default way to reference a record within Linz.

You create Models in the `model` directory; one file per model. The file should have the following basic structure:

person.js:

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

// Create a new mongoose schema.
var personSchema = new linz.mongoose.Schema({
  name:  String,
  email: String
});

// Add the Linz formtools plugin.
personSchema.plugin(linz.formtools.plugins.document, {
  model: {
    label: 'Person',
    description: 'A person.',
    title: 'name'
```

```
  },
  labels: {
    name: 'Name',
    email: 'Email'
  },
  list: {
    fields: {
      name: true,
      email: true
    }
  },
  form: {
    name: {
      fieldset: 'Details',
      helpText: 'The users full name.'
    },
    email: {
      fieldset: 'Details'
    }
  },
  overview: {
    summary: {
      fields: {
        name: {
          renderer: linz.formtools.cellRenderers.defaultRenderer
        },
        email: {
          renderer: linz.formtools.cellRenderers.defaultRenderer
        }
      }
    }
  },
  fields: {
    usePublishingDate: false,
    usePublishingStatus: false
  }
});

var person = module.exports = linz.mongoose.model('person', personSchema);
```

The file is broken down in the following parts:

- You `require` Linz, as you'll need to register the model with Linz.
- Create a standard Mongoose schema.
- Use the `linz.formtools.plugins.document` Mongoose plugin to register the model with Linz, passing in an object containing Linz's model DSL.
- Create a Mongoose model from the schema, and `export` it.

4.1 Mongoose schemas

Linz works directly with Mongoose schemas. Anything you can do with a Mongoose schema is acceptable to Linz.

4.2 Model DSL

Linz uses a Model DSL, which is an object that can be used to describe your model. Linz will use this information to scaffold user interfaces for you. The Model DSL contains six main parts:

- `model` contains basic information such as the `title` field, `label` and `description` of the model.
- `labels` contains human friendly versions of your model's properties, keyed by the property name.
- `list` contains information used to scaffold the list displaying model records.
- `form` contains information used to scaffold the edit handler for a model record.
- `overview` contains information used to scaffold the overview for a model record.
- `fields` contains directives to enable/disable fields that Linz automatically adds to models.
- `permissions` is a function used to limit access to a model.

You supply the DSL to Linz in the form of an object, to the `linz.formtools.plugins.document` Mongoose plugin:

```
personSchema.plugin(linz.formtools.plugins.document, {
  model: {
    // ...
  },
  labels: {
    // ...
  },
  list: {
    // ...
  },
  form: {
    // ...
  },
  overview: {
    // ...
  },
  fields: {
    // ...
  },
  permissions: function () {
  }
});
```

4.2.1 Models model DSL

The `model` keys value should be an object with three keys:

- `title` is required, unless you have a `title` field in your schema. If not, you should reference another field in your schema. This field will be used to derive the *title* for the record, and label for the field.
- `label` should be a singular noun describing the model.
- `description` should be a short sentence describing the noun.

The `label` is used in many places and is automatically pluralized based on the usage context. The `description` is only used on the Models index within Linz.

For example:

```
model: {
  label: 'Person',
  description: 'A person.',
  title: 'name'
}
```

4.2.2 Models label DSL

The label DSL is used to provide a label and description for the model.

The `labels` keys value should be an object, keyed by field names and strings of the human friendly versions of your field names.

For example:

```
labels: {
  name: 'Name',
  email: 'Email'
}
```

You can customize the labels for the default `dateModified` and `dateCreated` using this object. You can also supply the key `title` with a value that should be used for the label of the record's title.

4.2.3 Models list DSL

The list DSL is used to customize the model index that is generated for each model.

The `list` keys value should be an Object, containing the following top-level keys:

- `actions`
- `fields`
- `sortBy`
- `toolbarItems`
- `showSummary`
- `filters`
- `paging`
- `groupActions`
- `recordActions`
- `export`

These allow you to describe how the model index should function. The list DSL is discussed in more detail in *List DSL*.

Models list DSL function

The `list` keys value can also be a function. It should be a function with the following signature:

```
function listDSL (req, callback) {
```

For example:

```
{
  list: (req, callback) => callback(null, {
    fields: {...}
  })
}
```

The function receives a HTTP request object, which provides lots of flexibility to alter the DSL object returned based on the user making the request, and the model record itself.

4.2.4 Models form DSL

The form DSL is used to customize the model record create and edit pages.

The `form` keys value should be an Object, keyed by field names of the model, in the order you'd like each field's edit control rendered. For example:

```
form: {
  name: {
    fieldset: 'Details',
    helpText: 'The users full name.'
  },
  email: {
    fieldset: 'Details'
  }
}
```

This will generate a form with two fields that you can provide data for. Both fields will appear in the *Details* fieldset, in the order `name` and then `email`.

Each field object can contain the following keys:

- `label`
- `placeholder`
- `helpText`
- `type`
- `default`
- `list`
- `visible`
- `disabled`
- `fieldset`
- `widget`
- `required`
- `query`
- `transform`
- `transpose`
- `schema`
- `relationship`

These allow you to describe how the create and edit forms should function. The form DSL is discussed in more detail in *Form DSL*.

Models form DSL function

The `form` keys value can also be a function. It should be a function with the following signature:

```
function formDSL (req, callback) {
```

For example:

```
{
  form: (req, callback) => callback(null, {
    name: {...}
  })
}
```

The function receives a HTTP request object, which provides lots of flexibility to alter the DSL object returned based on the user making the request, and the model record itself.

4.2.5 Model permissions

Model permissions is an in-depth topic and should be considered amongst other permission capabilities. Read more about *Permissions*.

4.3 Model statics, virtuals and methods

When working with models, Linz makes use of specific Mongoose statics, virtuals and methods if they've been provided.

The following documents them, and their functionality.

4.3.1 listQuery static

You can create a Mongoose static called `listQuery` for a model with the following signature:

```
function listQuery (req, query, callback)
```

If found, Linz will execute this function with `req` and a Mongoose query before executing it, when retrieving data for the model list view. This provides an opportunity to customise the query before execution.

For example, if you'd like to return more fields from MongoDB than those listed in `list.fields` you can do it here:

```
model.static.listQuery = (req, query, callback) => callback(null, query.select(
↪'anotherField anotherOne'));
```

4.3.2 canDelete method

You can create a Mongoose method called `canDelete` for a model, with the following signature:

```
function canDelete (req, callback)
```

If found, Linz will execute this function before rendering the Model index page. This provides an opportunity to customise the delete record action. Because it is a Mongoose method, inside the function `this` is scoped to the record itself.

The callback has the following signature `callback (err, isEnabled, message)`. `isEnabled` should be a boolean; `true` to enable the delete action, `false` to disable it. If it is disabled, you can use `message` to provide a message that will be displayed to the user if they click on the delete button.

4.3.3 canEdit method

You can create a Mongoose method called `canEdit` for a model, with the following signature:

```
function canEdit (req, callback)
```

If found, Linz will execute this function before rendering the Model index page. This provides an opportunity to customise the edit record action. Because it is a Mongoose method, inside the function `this` is scoped to the record itself.

The callback has the following signature `callback (err, isEnabled, message)`. `isEnabled` should be a boolean; `true` to enable the edit action, `false` to disable it. If it is disabled, you can use `message` to provide a message that will be displayed to the user if they click on the edit button.

CHAPTER 5

Permissions

Linz has a unique permissions model, mostly due to the fact that it assumes nothing (well, nothing that you can't alter anyway) about your user model; only that you have one.

Many frameworks define a user model that you must adhere to. Linz doesn't. This provides an opportunity for a simplified yet highly flexible permissions model.

Permissions can be provided for both models and configs.

There are a few contexts you should be aware of.

The full scope of contexts are:

- **In the context of all models:**
  - `models.canList`
- **In the context of a particular model:**
  - `model.canCreate`
  - `model.canDelete`
  - `model.canEdit`
  - `model.canList`
  - `model.canView`
- **In the context of all configs:**
  - `configs.canList`
- **In the context of a particular config:**
  - `config.canEdit`
  - `config.canList`
  - `config.canReset`
  - `config.canView`

- config.canView

Linz enforces permissions in two places:

- The UI
- A route execution

Linz will not render buttons, links to or actions for functionality that a user doesn't have access to. Routes are completely protected. So even if a route was discovered, a user without permissions would not be able to resolve it.

5.1 Default permissions

This is Linz's default permissions implementation:

```
function (user, context, permission, callback) {
  return callback(true);
}
```

In short, there are no permissions.

5.2 Global permissions

Linz implementations can provide a function (in the `options` object when initializing Linz) called `permissions`. It should have the signature:

```
function permission (user, context, permission, callback)
```

This function will be called whenever Linz is evaluating the `models.canList` and `configs.canList`. Most commonly when generating navigation for a user, but also on the models list and configs list pages.

The `user` is the user making the request for which permissions are being sought.

The context will be either a string, or an object. If it is a string, it will be either:

```
// In the context of all models
'models'

// In the context of all configs
'configs'

// In the context of a particular model
{
  'type': 'model',
  'model': 'modelName'
}

// In the context of a particular config
{
  'type': 'config',
  'config': 'configName'
}
```

The `permission` will be one of the following strings:

- `canCreate`

- canDelete
- canEdit
- canList
- canReset (configs only)
- canView

The callback accepts the following signature:

```
function callback (result)
```

result is a boolean. Please note, this is different from the standard Node.js callback signature of function callback (err, result). You should design your function so that it returns false in the event of an error and logs the error for a post-mortem.

Throwing errors and failing at the point of checking permissions would not be a good look for anyone, hence the design to not provide this capability. This is something that needs to be handled by a developer.

5.3 Model and config permissions function

Determining permissions for models and configs is more contextually sensitive. To do this, when defining a model or config, you can also provide a permissions key.

The key can have a value of either an object or a function. If an object is provided, it is used directly. If a function is provided, you have the benefit of knowing which user the permissions are being requested for. A function should have the following signature:

```
function modelPermission (user, callback)
```

The callback accepts the following signature:

```
function callback (err, result)
```

err should be null if no error occurred. If an error has occurred, you can return it to the callback which will then default the result to false. result should be an object.

The result object should contain, optionally, the following keys with boolean values:

- canEdit
- canDelete
- canList
- canCreate
- canView

Each key is optional, and defaults to true if not provided. Linz evaluates the values with the === operator so an explicit false must be provided to limit permissions.

CHAPTER 6

Plugins

Linz comes with a number of useful mongoose plugins.

6.1 model.queryPlugin

The query plugin extends the mongoose `find` and `findOne` static methods through the methods `findDocuments` and `findOneDocument`.

`findDocuments` accepts all mongoose query options within a single options object. Note the defaults in the example.

```
Model.findDocuments({
    filter: {},
    lean: true,
    limit: 10,
    projection: '',
})
```

`findOneDocument` accepts all mongoose query options within a single options object. Note the defaults in the example.

```
Model.findOneDocument({
    filter: {},
    lean: true,
    projection: '',
})
```

CHAPTER 7

List DSL

The Models list DSL is used to customise the model index that is generated for each model. The list DSL has quite a few options, as the model index is highly customizable.

`list` should be an object, containing the following top-level keys:

- `actions`
- `export`
- `fields`
- `filters`
- `groupActions`
- `help`
- `paging`
- `recordActions`
- `showSummary`
- `sortBy`
- `toolbarItems`

These allow you to describe how the model index should function.

7.1 list.actions

`list.actions` should be an Array of Objects. Each object describes an action that a user can make, at the model level. Each action should be an Object with the following keys:

- `label` is the name of the action.
- `action` is the last portion of a URL, which is used to perform the action.

- `modal` optionally render the results in a modal view.

For example:

```
actions: [
  {
    label: 'Import people',
    action: 'import-from-csv',
    modal: true
  }
]
```

This will generate a button, on the model index, next to the model label. Multiple actions will produce a button titled *Actions* with a drop-down list attached to it, containing all possible actions.

The evaluated string `/{linz-admin-path}/model/{model-name}/action/{action.action}` will be prefixed to the value provided for `action` to generate a URL, for example `/admin/model/person/import-from-csv`. It is the developers responsibility to mount the `GET` route using Express, and respond to it accordingly.

The actions will be rendered in the order they're provided.

If using a modal, make sure the HTML returned from the route starts with `<div class="modal-dialog"><div class="modal-content"></div></div>`.

7.2 list.export

`list.export` is used to denote that a particular model is exportable. Linz takes care of the exporting for you, unless you want to provide a custom action to handle it yourself.

When a user clicks on an export, they'll be provided a pop-up modal asking them to choose and order the fields they'd like to export.

`list.export` should be an Array of Objects. Each object describes an export option, for example:

```
export: [
  {
    label: 'Choose fields to export',
    exclusions: 'dateModified,dateCreated'
  }
]
```

Each object should contain the following keys:

- `label` which is the name of the export.
- `exclusions` which is a list of fields that can't be exported.

If you'd like to provide your own export route, you can. Replace the `exclusions` key with an `action` key that works the same as *list.actions*. Rather than a modal, a request to that route will be made. You're responsible for mounting a `GET` route in Express to respond to it.

7.3 list.fields

`list.fields` is used to customize the fields that appear in the listing on the model index.

`list.fields` should be an Object, keyed by each field in your model. The value for each key should be `true` to include the field or `false` to exclude the field. For example:

```
fields: {
  name: true,
  username: true
}
```

Linz will convert the above into the following:

```
fields: {
  name: {
    label: 'Name',
    renderer: linz.formtools.cellRenderers.default
  },
  username: {
    label: 'Username',
    renderer: linz.formtools.cellRenderers.default
  }
}
```

If you like, you can pass an object rather than the boolean. This also allows you to customize the cell renderer used to display the data within the column.

If you provide a `label`, it will override what is defined in the *Models label DSL*.

The fields will be rendered in the order they're provided.

7.4 list.filters

`list.filters` can be used to include filters which will alter the data included in the dataset for a particular model. Filters can contain a custom user interface, but Linz comes with a standard set of filters.

`list.filters` should be an object, keyed by each field in your model. Each object must contain a filter, which should be an object adhering to the Linz model filter DSL. For example:

```
filters: {
  dateModified: {
    alwaysOn: true,
    filter: linz.formtools.filters.dateRange,
  }
}
```

The above will allow your model to be filtered by a date range filter, on the `dateModified` property.

Each filter, keyed by the field name, can have the following keys:

- `alwaysOn` will ensure that the filter is always rendered in the list view.
- `default` allows you to provide a default value for the filter. It only takes affect when using `alwaysOn`.
- `filter` this is optional, but allows you to specify a filter and should point to a Linz filter, or your own custom one.
- `once` will ensure that a user can only add that filter once (works well with the `boolean` filter).

Note: Be aware of the `default` values. Because of Linz's internal query structure most filters will need to provide the `default` value as an array, but there are some exceptions.

Below is an example of the `default` data type for each filter:

- `dateRange`: `{ dateFrom: [ '2017-10-15' ], dateTo: [ '2017-10-28' ] }`
- `date`: `['2017-10-01']`
- `boolean`: `true`
- `default`, `fulltext`, `list`: `['string']`
- `number`: `[4]`

See also:

View the complete list of Linz filters.

7.5 list.groupActions

`list.groupActions` can be used to define certain actions that are only available once a subset of data has been chosen.

Each record displayed on a model index has a checkbox, checking two or more records creates a group. If `groupActions` have been defined for that model, those actions will become choosable by the user.

`list.groupActions` should be an Array of Objects. Each object describes an action that a user can make, and the object takes on the same form as those described in *list.actions*.

You're responsible for mounting a `GET` route in Express to respond to it.

7.6 list.help

The `list.help` key can be used to provide information for a particular model. The information will appear in a Bootstrap popover.

The `list.help` key accepts either `false`, or a Bootstrap popovers options object.

7.7 list.paging

`list.paging` can be used to customise the paging controls for the model index. Paging controls will only be shown when the number of results for a model index, are greater than the per page total.

`list.paging` should be an Object, with the following keys:

- `active` is an optional Boolean used to turn paging on or off. It defaults to `true`.
- `size` is the default page size. It defaults to `20`.
- `sizes` is an Array of the page sizes available for a user to choose from on the model index. It defaults to `[20, 50, 100, 200]`.

For example:

```
paging: {
  active: true,
  size: 50,
  sizes: [50, 100, 150, 200]
}
```

If you don't provide a paging object it defaults to:

```
paging: {
  active: true,
  size: 20,
  sizes: [20, 500, 100, 200]
}
```

7.8 list.recordActions

`list.recordActions` can be used to customise record specific actions. These are actions that act upon a specific model record. The actions appear as buttons for each record in a model list. The buttons can either appear in a drop-down list, or next to the edit and delete buttons for the record.

`list.recordActions` should be an Array of Objects. Each object describes an action that a user can make, specific to the record, and the object takes on the same form as those described in *list.actions*.

`list.recordActions` can have an optional key `type` and when set to `primary`, the action will be renderered next to the edit and delete buttons for the record (i.e. not within the dropdown). You can also supply a key `icon`, which if supplied, will be used rather than a label for the button. The value for `icon` should correspond with name of a Bootstrap glyphicon.

`list.recordActions` can also accept a function, as the value to a `disabled` property. If provided, the function will be excuted with the following signature `disabled (record, callback)`. The callback has the following signature `callback (error, isDisabled, message)`. `isDisabled` should be a boolean. `true` to disable the record action, `false` to enable it and you can provide a message if the action is to be disabled.

You're responsible for mounting a `GET` route in Express to respond to it.

7.9 list.showSummary

`list.showSummary` can be used to include or exclude the paging controls from a model index.

`list.showSummary` expects a boolean. Truthy/falsy values will also be interpreted, for example:

```
showSummary: true
```

7.10 list.sortBy

`list.sortBy` is used to customise the sort field(s) which the data in the model index will be retrieved with.

`list.sortBy` should be Array of field names, for example:

```
sortBy: ['name', 'username']
```

This Array will be used to populate a drop-down list on the model index. The user can choose an option from the drop-down to sort the list with.

7.11 list.toolbarItems

`list.toolbarItems` can be used to provide completely customised content on the toolbar of a model index. The toolbar on the model index sits directly to the right of the Model label, and includes action buttons and drop-downs.

`list.toolbarItems` should be an Array of Objects. Each object should provide a `render` key with the value of a Function. The function will be executed to retrieve HTML to be placed within the toolbar. The function will be provided the request *req*, the response object *res* and callback function which should be executed with the HTML. The callback function has the signature `callback(err, html)` For example:

```
toolbarItems: [
  {
    renderer: function (req, res, cb) {

      let locals = {};
      return cb(null, templates.render('toolbarItems', locals));

    }
  }
]
```

CHAPTER 8

Form DSL

The Models form DSL is used to customise the create and edit forms that are generated for each model. The form DSL has quite a few options as the model create and edit forms are highly customizable.

The form DSL is used to construct create and edit form controls (for example checkboxes, or text inputs) for a model record. Each key in the `form` object represents one of your model's fields.

The type of form control used for each field can be defined explicitly, or determined by Linz (the default) based on the fields data type, as specificed when defining the field with Mongoose.

Each form control comes in the form of a widget, and can be explicitly altered by providing a different Linz widget, or creating your own widget.

`form` should be an object. It should contain a key, labelled with the name of the model field you're providing information for.

For example, if you had a model with the fields `name` and `email` your `form` DSL might look like:

```
form: {
  name: {
    // configure the edit widget for the name field
  },
  email: {
    // configure the edit widget for the name field
  }
}
```

Each field object can contain the following top-level keys:

- `label`
- `placeholder`
- `helpText`
- `type`
- `default`
- `list`

- visible
- disabled
- fieldset
- widget
- required
- query
- transform
- transpose

These allow you to describe how the model create and edit forms should function.

8.1 Specialized contexts

There are two specialized contexts in which the form DSL operates:

- When creating a model
- When editing a model

From time to time, you'll want to have different settings for one field, based on the context. Linz supports this through use of create and edit keys. Each of the above top-level keys can also be provided as a child of either create and edit. For example:

```
form: {
  username: {
    create: {
      label: 'Create a username',
      helpText: 'You can\'t change this later on, so choose wisely.'
    },
    edit: {
      label: 'The person\'s username',
      disabled: true,
      helpText: 'Once created, you can\'t edit the username.'
    }
  }
}
```

You can also use a combination of the default context and the specialized contexts create and edit contexts, for example:

```
form: {
  username: {
    label: 'The person\'s username',
    edit: {
      label: 'Uneditable username'
    }
  }
}
```

On the create form, the label for the username field will be *The person's username*, but *Uneditable username* on the edit form.

The specialized create and edit contexts always supersede the default context.

8.2 {field-name}.label

The `label` property is optional. If not provided, it takes the label from the *Models label DSL*. If a label hasn't been provided for that particular model field, it simply shows the name of the field itself.

The label property gives you an opportunity to customize it explicitly for the create and edit views.

8.3 {field-name}.placeholder

When you have the field of an appropriate type (such as text field), you can define the `placeholder` which sets the content of the HTML's `<input>` tag `placeholder` attribute.

When used in conjunction with a `ref` field, it can be used to create optional references. For example, a `select` in which the first `option` has no `value` but contains the `placeholder` value as the label.

8.4 {field-name}.helpText

The `helpText` property can be used to supply additional text that sits below the form input control, providing contextual information to the user filling out the form.

8.5 {field-name}.type

The `type` property is intended to help Linz with two things:

- Manage the data that the field contains in an appropriate manner.
- To determine which widget to use if the `widget` property wasn't provided.

`type` accepts the following strings:

- `array` to render checkboxes for multiple select.
- `boolean` to render radio inputs.
- `date` to render a date input.
- `datetime` to render a datetime input.
- `datetimeLocal` to render a datetime-local input.
- `digit` to render a text input with a regex of `[0-9]*`.
- `documentarray` to render a custom control to manage multiple sub-documents.
- `email` to render an email input.
- `enum` to render a select input.
- `hidden` to render a hidden input.
- `number` to render a text input with a regex of `[0-9,.]*`.
- `password` to render a password input.
- `string` to render a text input.
- `tel` to render a tel input with a regex of `^[0-9 +]+$`.

- `text` to render a text input.
- `url` to render a url input.

The default widget, and the widget for all other types is the text widget.

8.6 {field-name}.default

The `default` property can be supplied to define the default value of the field. The default if provided, will be used when a field has no value.

If the `default` property is not provided, Linz will fallback to the `default` value as provided when defining the *Mongoose schemas*.

8.7 {field-name}.list

The `list` property is a special property for use with the `enum` type. It is used to provide all values from which a list field value can be derived.

Please bear in mind, that the `list` property is not involved in Mongoose validation.

The `list` property can either be an array of strings, or an array of objects.

For example, an array of strings:

```
list: [ 'Dog', 'Cat', 'Sheep' ]
```

If an array of objects is supplied, it must be in the format:

```
form: {
  sounds: {
    list: [
      {
        label: 'Dog',
        value: 'woof.mp3'
      },
      {
        label: 'Cat',
        value: 'meow.mp3'
      },
      {
        label: 'Sheep',
        value: 'baa.mp3'
      }
    ]
  }
}
```

There is also a more advanced use case in which you can provide a function which Linz will execute. This will allow you to generate at run time rather than start time, after Linz has been initialized:

```
form: {
  sounds: {
    list: function (cb) {
      return cb(null, {
        label: 'Dog',
```

```
        value: 'woof.mp3'
      },
      {
        label: 'Cat',
        value: 'meow.mp3'
      },
      {
        label: 'Sheep',
        value: 'baa.mp3'
      });
    }
  }
}
```

8.8 {field-name}.visible

The boolean `visible` property can be set to a value of `false` to stop the field from being rendered on the form.

8.9 {field-name}.disabled

The boolean `disabled` property can be set to a value of `true` to render the input field, with a disabled attribute.

8.10 {field-name}.fieldset

The `fieldset` property should be supplied to control which fields are grouped together under the same fieldset.

The `fieldset` property should be human readable, such as:

```
form: {
  username: {
    fieldset: 'User access details'
  }
}
```

8.11 {field-name}.widget

The `widget` property can be set to one of the many built-in Linz widgets. For example:

```
form: {
  sounds: {
    widget: linz.formtools.widget.multipleSelect()
    list: [
      {
        name: 'Dog',
        value: 'woof.mp3'
      },
      {
        name: 'Cat',
        value: 'meow.mp3'
```

```
      },
      {
        name: 'Sheep',
        value: 'baa.mp3'
      }
    ]
  }
}
```

8.12 {field-name}.required

The boolean `required` property can be set to `true` to require that a field has a value before the form can be saved (using client-side) validation.

8.13 {field-name}.query

The `query` property can be used to directly alter the Mongoose query object that is generated while querying the database for records to display.

`query` should be an object with the following keys:

- `filter`
- `sort`
- `select`
- `label`

8.14 {field-name}.transform

The `transform` property will accept a function with the signature:

```
transform (field, 'beforeSave', form, user)
```

If provided, it will be executed before a record is saved to the database. It is useful if you need to manipulate client side data before it is stored in the database.

In some instances, client side data requirements are different from that of data storage requirements. `transform` in combination with `transpose` can be used effectively to manage these scenarios.

8.15 {field-name}.transpose

The `transpose` property will accept a function with the signature:

```
transpose (field, record)
```

If provided, it will be executed before a field's value is rendered to a form. It is useful if you'd like to manipulate the server-side data that is rendered to a form.

In some instances, data storage requirements are different form that of client side data requirements. `transpose` in combination with `transform` can be used effectively to manage these scenarios.

CHAPTER 9

Notifications

Linz provides the ability to pop up notifications, into a Noty message.

It works well when paired with model, group, overview and record actions that perform a task, and then redirect back to the original page.

When using Linz notifications in this manner, you must make use of connect-flash like so:

```
// Perform task.

// Create the notification.
req.flash('linz-notification', linz.api.view.notification({ text: 'Notification␣
↪message', type: 'success' }));

// Redirect the user back.
return res.redirect('back');
```

When the page is rendered, a notification will appear, informing the user that the action they took was successful.

9.1 API

There are two APIs for using notifications:

- `req.linz.notifications`
- `req.flash`

9.2 req.linz.notifications

Before rendering a page, at some point in the route execution middlware, you can populate the `req.linz.notifications` array with any Noty objects you'd like to be shown. For example:

```
app.use('/url', (req, res, next) => {

  req.linz.notifications.push(linz.api.views.notification({ text: 'A message here.' }
↪));

  return next();

});
```

Once the page is rendered, Linz will pick up on this notification and display it for the user.

9.3 req.flash

As described above, using the `req.flash` API for notifications is a handy way to provide the user information about the state of the action they've just performed. However, it should always be used in conjunction with `res.redirect('back')`.

To use this API, use `req.flash` like so:

```
req.flash('linz-notification', linz.api.views.notification({ text: 'Message here.' }
↪));
```

The first parameter passed to `req.flash` must be the string `'linz-notification'`, otherwise Linz will ignore it. The second parameter passed must be a Noty options object. Linz provides a handy API (*views.notification(noty)*) defaulting some of the options.

CHAPTER 10

Request namespace

Linz adds to the Express `req` object, an object which you can use to access Linz information about the incoming request.

The Linz namespace exists at:

```
req.linz
```

And is a copy of the object you receive when requiring Linz, for example `require('linz')`.

It has the keys:

- `notifications` which is an array of notifications Linz will display.
- `cache` which is an internal cache that Linz uses.

Depending on which view is currently being requested, you'll also get extra information.

The Linz namespace can be used whenever Linz passes you `req` and becomes a very handy API to get more infromation about the request currently being served.

10.1 Model form

The model form, both create and edit views, also recieve:

- `model` which is a reference to the current model, the basic Mongoose version of the model.
- `model.linz` which is a reference to the current model, which extra Linz-specific information included.
- `model.linz.form` which is a reference to the model form DSL.

10.2 Model list

The model list view also recieves:

- `model` which is a reference to the current model, the basic Mongoose version of the model.
- `model.linz` which is a reference to the current model, which extra Linz-specific information included.
- `model.linz.list` which is a reference to the model List DSL.

10.3 Model overview

The model overview view also recieves:

- `model` which is a reference to the current model, the basic Mongoose version of the model.
- `model.linz` which is a reference to the current model, which extra Linz-specific information included.
- `model.linz.overview` which is a reference to the model overview DSL.

CHAPTER 11

Cell renderers

Linz tries to provide as many customisation options as possible. One of those is in the form of a what we call a cell renderer.

Cell renderers can be used within record overviews and model indexes. They're used to represent data to the user in a human friendly way.

You can do many things with cell renderers that will improve the user experience. For example, you could take latitude and longitude values and render a map, providing visual context about location information specific to the record.

11.1 Built-in cell renderers

There are already many built-in cell renderers. They can all be accessed in the following namespace `linz.formtools.cellRenderers`.

The following shows how to define a specific cell renderer for a list field:

```
list: {
  fields: {
    websiteUrl: {
      label: 'Website url',
      renderer: linz.formtools.cellRenderers.url
    }
  }
}
```

The following provides a description of each built-in cell renderer:

- `date` used with `date` field types to render a date, as per the `date format` setting.
- `datetime` used with `datetime` field types to render a datetime, as per the `datetime format` setting.
- `localDate` used with `datetime` field types to render a `<time>` tag as per the `date format` setting.
- `datetimeLocal` used with `datetime-local` field types to render a `<time>` tag as per the `datetime format` setting.

- `overviewLink` can be used to provide a link in the list, to the overview for a particular record.
- `array` can be used to format an array in the format `value 1, value 2, value 3`.
- `boolean` can be used to format a boolean in the format `Yes` or `No`.
- `reference` can be used to render the title for a `ref` field type.
- `url` can be used with `url` field types to render an `<a>` tag linking to the URL stored as the value of the field.
- `documentarray` can be used with an embedded document to render a table showing embedded documents.
- `text` can be used to render text, or any value as it is.
- `default` is used by Linz as the default cell renderer if a specific type can't be matched. It attempts to support arrays, dates, numbers, booleans and url field types.

11.2 Custom cell renderers

You can quite easily create and use your own cell renderer. All cell renderers must have the same signature:

```
function renderer (value, record, fieldName, model, callback)
```

The `value` is the value that needs to be rendered. The `record` is a copy of the entire record that the `value` belongs to. The `fieldName` is the name of the field that is being rendered. The `model` is a reference to the model that the `record` belongs to.

The `callback` is a function that accepts the standard Node.js callback signature:

```
function callback (err, result)
```

The `result` should be HTML. The HTML will be used directly, without manipulation. As it is HTML, you can provide inline JavaScript and CSS as required to add functionality your cell renderer.

The following is an example of a cell renderer that will look up data for a reference field and render the title:

```
function renderReference (val, record, fieldName, model, callback) {

  // Make sure we have the neccessary value, without erroring.
  if (!val || typeof val === 'string' || typeof val === 'number') {
    return callback(null, val);
  }

  // Retrieve the related documents title.
  linz.mongoose.models[val.ref].findById(val._id, (err, doc) => {

    if (err) {
      return callback(err);
    }

    return callback(null, (doc && doc.title) ? doc.title : `${val} (missing)`);

  });

};
```

CHAPTER 12

Forgot password process

Linz has the capability to support a *forgot password* process. It can be used to allow the user to reset their password.

If enabled, it will render a *Forgot your password?* link on the log in page, which will facilitate the ability for a user to reset their password. It uses an email as proof of user record ownership. If the user can access a link sent to an email identified with a user record, then they can reset the password.

To enable this process, you need to:

- Have a user model that stores an email address.
- Define a `sendPasswordResetEmail` Mongoose static on your user model.
- Define a `verifyPasswordResetHash` Mongoose method on your user model.
- Define an `updatePassword` Mongoose static on your user model.

The process works as follows:

- User clicks the *Forgot your password?* link on the log in page.
- The user is directed to the *Forgot your password* page, and prompted to enter their email address.
- The `sendPasswordResetEmail` static is executed with the email address they provided.
- The `sendPasswordResetEmail` static should generate a unique hash for the user, and send an email to the user containing a link to reset their password.
- The user will receive the email, and click on the link.
- The link will be verified by executing the `verifyPasswordResetHash` method.
- If the password reset hash can be be verified, the user will be provided the opportunity to enter a new password meeting the conditions of the `admin password pattern` setting.
- The new password will be provided to the `updatePassword` static to store the updated password against the user record.

More succintly:

1. Send a password reset email.

2. Verify ownership of the email.
3. Collect a new password and update their user record.

12.1 Send a password reset email

This part of the process entails:

- Retreiving a user record based on an email address.
- Generating a unique hash for the user record.
- Creating a link for the user to continue the process.
- Sending an email to the email address provided.

These actions should take place within the `sendPasswordResetEmail` executed by Linz.

12.1.1 The `sendPasswordResetEmail` static

The `sendPasswordResetEmail` static should have the following signature:

```
function sendPasswordResetEmail (userEmail, req, res, callback)
```

It needs to be a Mongoose static on your user model.

`userEmail` is the email address provided by the user who is trying to reset their password, `req` is the current request object, `res` is the current response object and `callback` is the function to execute when you've completed the neccessary steps.

The `callback` accepts the standard Node.js signature:

```
function callback (err)
```

If an `Error` is provided, Linz will render the error, otherwise it will consider the process complete.

12.1.2 Retrieving a user record based on an email address

Use the `userEmail` argument to search your user model for a corresponding record. If a record can't be found, return an `Error` to the `callback`.

Make sure you take into consideration the following scenarios:

- Multiple user records associated with the same email address.
- No user record associated with the email address.

12.1.3 Genearing a unique hash for the user record

Once you have the user record, generate a unique hash for the user record. We recommend including the `username`, `_id`, `email` and `dateModified`.

The hash you generate must be verifyable by generating the same hash, at a later time, with the same information in the database (i.e. `username`, `_id`, `email` and `dateModified`).

A good Node.js package to consider to generate a hash is the bcrypt.js package.

12.1.4 Creating a link to verify email address ownership

Once you have the unique hash, and the records `_id` value you can use `linz.api.url.getAdminPasswordResetLink(id, hash)` to generate a url. Pass in the `_id` and hash and Linz will safely add those to the url it returns.

12.1.5 Send an email

Once you have the link, you simply need to send it to the email address with instructions on what to do next; click on the link.

This is something you'll have to implement yourself. Linz does not provide any capabilities to send emails. Linz is based on Express though, so you have all of it's templating capabilities at hand. See using template engines with Express.

12.2 Verifying ownership of the email address

This part of the process involves verifying ownership of the email address. The user will receive the email, and click on the link. We want to make sure the link hasn't been tampered with and that we can generate the same hash that was provided in the link.

Linz will retrieve the hash from the url and pass it to the `verifyPasswordResetHash` method.

It must be a Mongoose method on your user model.

Your `verifyPasswordResetHash` Mongoose method should have the following signature:

```
function verifyPasswordResetHash (candidateHash, callback)
```

The `candidateHash` is the hash value that was retreived from the Url. The `callback` is a standard Node.js callback:

```
function callback (err, result)
```

The `result` should be a boolean value.

Your `verifyPasswordResetHash` method should go through the same process to create the hash as it did in the first process. It should then verify that the `candidateHash` is the same as your freshly generated hash using the data from your database.

If the `candidateHash` checks out and you can successfully match it, return `true` to the callback.

12.3 Updating the users password

If the `hash` was verified, the user is provided an opportunity to enter a new password. The new password must meet the requirements of the `admin password` setting.

The new password is provided to the `updatePassword` Mongoose static on your user model. The `updatePassword` static should have the following signature:

```
function updatePassword (id, newPassword, req, res, callback)
```

`id` is the `_id` of the user model record. `newPassword` is the new password provided by the user. `req` is the current request object. `res` is the current response object. `callback` is a standard Node.js callback:

```
function callback (err)
```

If an `Error` is provided, Linz will render the error, otherwise it will consider the process complete.

The user will be notified that their password has been updated, and prompted to log into Linz again.

CHAPTER 13

Getting started with Linz development

The linz-development repository is a complete environment for hacking on Linz. Using Vagrant, and a few commands, you'll have a complete development environment up and running in no time.

Visit the linz-development repository for more information on how to get started hacking on Linz.

13.1 A note on documentation

Documentation is now a primary concern for the project. All PRs should be accompanied with updated documentation that describe in detail how to use a new feature, new capability, updates to an existing DSL or a new DSL.

CHAPTER 14

mini-twitter

Mini Twitter is a complete working example of Linz. Head over to the mini-twitter GitHub repository to download mini-twitter and check out Linz.

CHAPTER 15

Linz definitions

The following are a list of words you'll see used many times in the Linz documentation. You can familiarize yourself with our terminology here.

DSL Domain specific language. Linz uses these frequently to take instruction as to how certain functionality should be scaffolded. There is a DSL for Models, Configurations and Schemas.