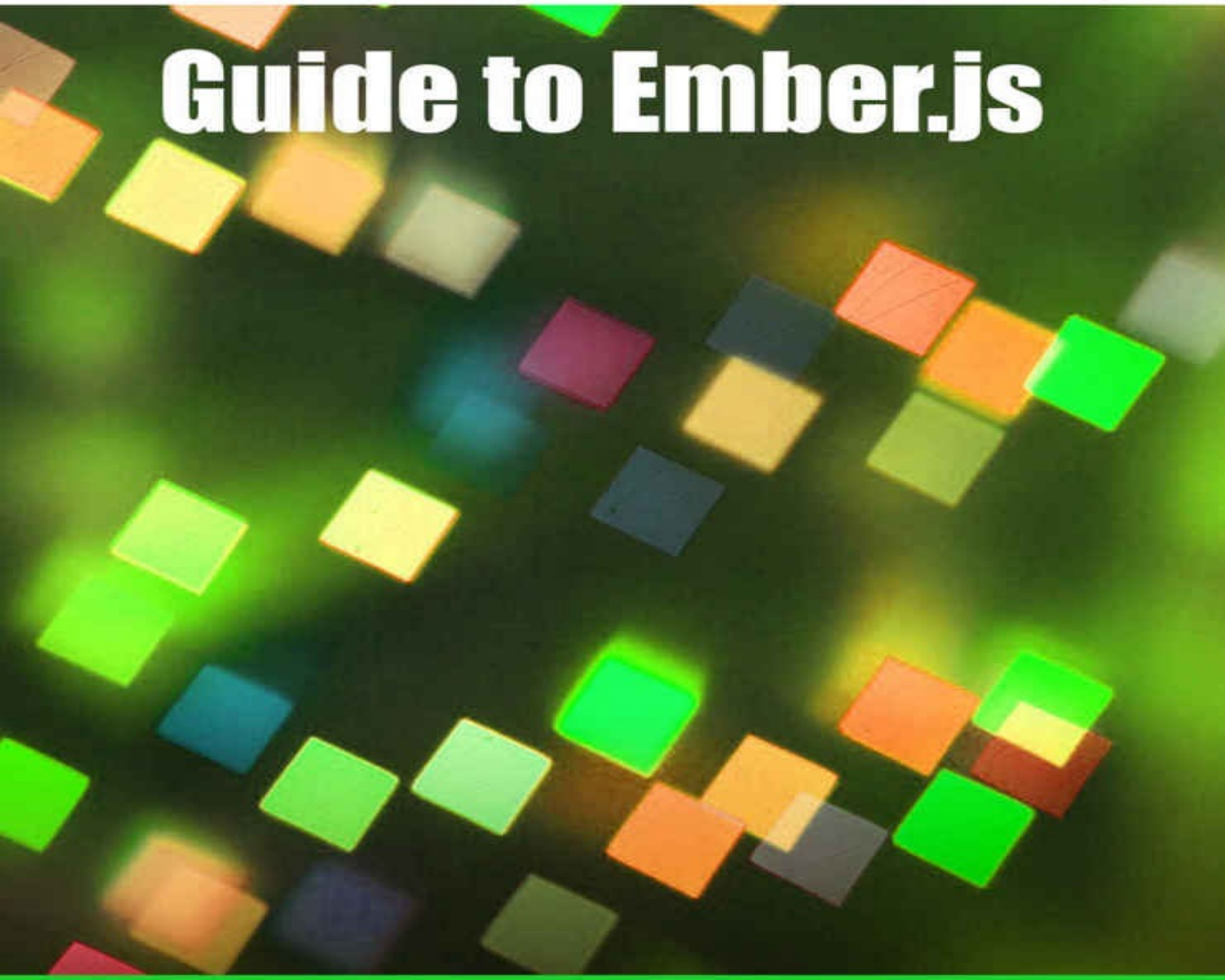
Web App Development Book

Guide to Ember.js



David V.

Web App Development Book

Guide to Ember.js

David V.

Table of Contents

Disclaimer

While all attempts have been made to verify the information provided in this book, the author does assume any responsibility for errors, omissions, or contrary interpretations of the subject matter contained within. The information provided in this book is for educational and entertainment purposes only. The reader is responsible for his or her own actions and the author does not accept any responsibilities for any liabilities or damages, real or perceived, resulting from the use of this information.

The trademarks that are used are without any consent, and the publication of the trademark is without permission or backing by the trademark owner. All trademarks and brands within this book are for clarifying purposes only and are the owned by the owners themselves, not affiliated with this document.

Introduction

There is an increase in the demand for web apps. Ember JS is a JavaScript which can assist in development of such apps. The framework is easy to learn, and especially for JavaScript experts. However, even if you are not good in JavaScript at all, you don’t have to worry, as the explanations in this book are very easy for you to understand. With this JavaScript, you can create web apps with very interactive user interfaces, and your users will definitely like your apps.

Chapter 1-Controllers

Controllers are much like components, and it is believed that these will be replaced by components in the future.

In most of the current apps, controllers are not used much. When they are used, they are tasked with the following responsibilities:

1. Used for maintaining state based on the route in use currently.

2. When the user actions are passing from a component to a route, they have to pass through a controller.

We need to demonstrate this by use of a route which will help us display a blog post.

The template will be bound to the following properties in the template: **<h1> {{model.title}}</h1> <h2>by {{model.author}}</h2> {{model.intro}}**

** **

<hr>

** {{model.body}}**

** **

As shown in the above example, we do not have specific properties or actions to display.

Suppose that we need to add a feature which will allow the user to toggle between the displays of your body sections. For us to implement this, we would modify the code to the following: **<h1>{{model.title}}</h1> <h2>by {{model.author}}</h2> {{model.intro}}**

** **

<hr>

{{#if isExpanded}}

<button {{action "toggleBody"}}>Hide the Body</button> {{model.body}}

** **

{{else}}

<button {{action "toggleBody"}}>Show the Body</button> {{/if}}

In the "*actions*" hook of our controller, we can define what our action will be

doing as we can do it with a component. This is shown in the code given below:

```
export default Ember.Controller.extend({

 actions: {

   toggleBody() {

     this.toggleProperty('isExpanded'); }

                                                    }

                                                  });
```

Management of Dependencies between Controllers

Sometimes, when we are nesting resources, we usually need to establish a connection between two controllers. Consider the router given below:

```
Router.map(function() {
 this.route('post', { path: '/p:post_id' }, function() {
   this.route('comments', { path: '/comments' }); });
});
```

On visiting the URL "*/posts/1/comments*," the model "*post*" will be loaded into the model of the PostController. However, some information related to it might be displayed in the template for "*comments.*"

For us to do this, we have to inject the "*pController*" into our "CommentsController." This is shown below:

```
export default Ember.Controller.extend({
 pController: Ember.inject.controller('post') });
```

Our comments will be in a position to access the "*PController,*" which is a read-only alias which can be used for reading the model from the controller. Consider the code given below:

```
export default Ember.Controller.extend({
```

pController: Ember.inject.controller('post'), post:

Ember.computed.reads('pController.model') });

Consider the next one for the .hbs file: **<h1>Comments for the {{post.title}}**

**</h1> **

{{#each model as |comment|}}

**
{{comment.text}} {{/each}}**

**
**

Chapter 2-Models

One of the ways to build web applications is to implement user interface components which are highly coupled to data fetching.

Consider a situation in which you are creating a blog app. If you are creating the admin part of the app, you may be forced to give the component the responsibility of fetching and storing the data. Consider the code given below:

```
<h1>Comments for {{post.title}}</h1> <ul>
 {{#each model as |comment|}}
   <li>{{comment.text}}</li> {{/each}}
</ul>
```

The list of drafts contained in the template of the component can then be shown. The following code demonstrates this:

```
<ul>
 {{#each drafts key="id" as |draft|}}
   <li>{{draft.title}}</li> {{/each}}
</ul>
```

Note that the app is made up of a number of components. You may want to have

all of the drafts displayed on another page. This may require copying and pasting the old code into the new component: **export default Ember.Component.extend({**

willRender() {

$.getJSON('/myDrafts').then(data => {

this.set('drafts', data);

});

}

});

The .hbs file for the button code should be as follows: **{{#link-to 'drafts' tagName="button"}}**

Drafts ({{drafts.length}})

{{/link-to}}

With the above, our app will make two requests which are separate but for the same information.

In Ember data, the representation of a model is done using a subclass of the model which defines the relationships, attributes, and the behavior of the data which is presented to the user. With models, one can define the kind of data

which the server will provide to us. Consider the example given below of a model named "*human*": **export default DS.Model.extend({**

fName: DS.attr('string'),

birthday: DS.attr('date')

});

A model can also describe the relationship between itself and the rest of the models. Consider the example given below, which shows how this can be done:

export default DS.Model.extend({

lItems: DS.hasMany('line-item')

});

The above code is for an order. The code for the "*line.js*" should be as follows:

export default DS.Model.extend({

order: DS.belongsTo('order')

});

You have to note that the models themselves do not have data, but they are used for definition of the attributes, relationships, and the behavior of specific instances referred to as "*records.*"

Creating Records

When you need to create records, you just have to call the method "*createRecord()"* on your store. Consider the code given below:

```
store.createRecord('post', {
 title: 'Ember is a Great Framework', body: 'Lorem ipsum'
});
```

The stored object will be available to the controllers and the routers by use of "*this.store."*

Updating Records

You can make changes to the Ember JS data records by setting the attribute which you are in need of changing. This is shown in the code given below:

```
this.store.findRecord('human', 1).then(function(myfunction) {
 // ...once the record has been loaded myfunction.set('fName', "John");
});
```

All Ember.js conveniences are always available for modification of attributes. This can be done as shown below: **human.incrementProperty('age'); // Happy birthday!**

Persisting Records In Ember JS, records are persisted on the basis of a per-instance. When you call the method "*save()*" on any instance of the DS.Model, a network request will be made.

By default, newly created records in EmberJS will be posted to their type URL in Ember data. This is shown in the following code: **var p = store.createRecord('post', {**

title: 'Ember is Great',

body: 'Lorem ipsum'

});

p.save(); // => POST to '/posts'

Records which exist on the backend will be updated by use of the HTTP PATCH verb. This is shown in the code given below: **store.findRecord('post', 1).then(function(p) {**

p.get('title'); // => "Ember JS is Great"

p.set('title', 'New post');

p.save(); // => PATCH to '*posts*1'

});

Consider the code given below:

human.get('isAdmin'); //=> **false human.get('hasDirtyAttributes');**
//=> **false human.set('isAdmin', true);**
human.get('hasDirtyAttributes'); //=> **true human.changedAttributes();**
//=> **{ isAdmin: [false, true] }**

One can tell if an outstanding record has some outstanding changes which have not yet been saved by checking on its property “hasDirtyAttributes.”

At the above point, you can just save the changes by calling the “*save()*” method or just roll them back. Consider the code given below:

human.get('hasDirtyAttributes'); //=> **true human.changedAttributes();**
//=> **{ isAdmin: [false, true] }**
human.rollbackAttributes();
human.get('hasDirtyAttributes'); //=> **false human.get('isAdmin');**
//=> **false human.changedAttributes();** //=> **{}**

Handling Validation Errors

In case your backend server returns validation errors once you try to save, the property "*errors"* of your model will havethese. The errors can be displayed as shown below: **{{#each p.errors.title as |err|}}**

** {{err.message}} {{/each}}**

{{#each p.errors.body as |err|}}

** {{err.message}} {{/each}}**

Promises The "*save()*" method will return a promise, and this will make it easy for us to handle scenarios of either success or failure. Consider the common pattern which is shown below: **var p = store.createRecord('post', {**

title: 'Ember is Great',

body: 'Lorem ipsum'

});

var self = this;

function transitionToPost(p) {

self.transitionToRoute('posts.show', p); }

function failure(reason) {

// handling the error

}

p.save().then(transitionToPost).catch(failure); // => POST to '/posts'

// => transitioning to posts.show route

Deleting Records Deletion of records can be done in the same manner as creating records. The method "*deleteRecord()*" can be called on the instance of the DS.Model. This will then flag the record as "*isDeleted.*" This should be followed by the calling of the method "*save()*" which will save the deletion of the record. Consider the code given below: **store.findRecord('post', 1).then(function(p) {**

p.deleteRecord(); p.get('isDeleted'); // => true p.save(); // => DELETE to *posts*1

}); // OR

store.findRecord('post', 2).then(function(post) {

p.destroyRecord(); // => DELETE to *posts*2

});

Relationships Ember has several built-in relationships which define how the models relate to each other.

One-to-One

If you need to declare this type of relationship between two models, use the code given below: Here is the code for the file "*myuser.js*": **export default DS.Model.extend({**

profile: DS.belongsTo('profile') });

The file "*profile.js*" should then be as follows: **export default DS.Model.extend({**

user: DS.belongsTo('myuser')

});

One-to-Many relationship

This type of relationship can be declared as follows between two models: Consider the code given below for the file "*myposts.js*": **export default**

DS.Model.extend({

comments: DS.hasMany('comment') });

Here should be the code for the file “our*comments.js”*: **export default**

DS.Model.extend({

post: DS.belongsTo('p')

});

Explicit Inverses Ember JS will do the best it can to discover the kind of relationship existing between components. Sometimes, you might have many different types of relationships for the same type.

Consider the code given below:

```
export default DS.Model.extend({
 firstPost: DS.belongsTo('post', { inverse: null }), secondPost:
DS.belongsTo('mypost'),
 redPost: DS.belongsTo('mypost'),
 bluePost: DS.belongsTo('mypost')
                                                    });
```

Here is the code for the file "*mypost.js*": **export default DS.Model.extend({**

```
 comments: DS.hasMany('comment', {
   inverse: 'redPost'
 })
                                                    });
```

That is how we have linked the two files together. Note that we have many relationships.

Reflexive Relationships

For a reflexive relationship to be defined, you should have the inverse relationship already defined. If you do not have the inverse relationship, you can set it to "*null.*"

Consider the example given below, which shows a reflexive inverse relationship:

```
export default DS.Model.extend({
 children: DS.hasMany('folder', { inverse: 'parent' }), parent:
DS.belongsTo('folder', { inverse: 'children' }) });
```

The example given below shows a one-to-one reflexive relationship:

```
export default DS.Model.extend({
 name: DS.attr('string'),
 bFriend: DS.belongsTo('user', { inverse: 'bFriend' }), });
```

A reflexive relationship without an inverse can be defined as follows:

```
export default DS.Model.extend({
 parent: DS.belongsTo('folder', { inverse: null }) });
```

Creating records

Suppose that we have two models, namely "*myposts*" and "*comments.*" These two are related as follows: mypsosts.js:

export default DS.Model.extend({

comments: DS.hasMany('comment')

});

The file "*comment.js*" is as follows: **export default DS.Model.extend({**

post: DS.belongsTo('post')

});

Once a user has commented on a post, a relationship has to be created between two records. The relationship "*belongsTo*" can simply be set to the comment as shown below: **let p = this.store.peekRecord('post', 1); let comm = this.store.createRecord('comment', {**

post: p

});

comm.save();

With the above code, a new comment will be created and then saved to the server. The post will also be updated by the Ember server so that the newly created comment can be included.

For us to link two records, we can also update the "*hasMany*" relationship for the post. This is shown in the code given below: **let p =**

this.store.peekRecord('post', 1); let comm =

this.store.createRecord('comment', {

});

p.get('comments').pushObject(comm);

comm.save();

In our above case, the relationship "*belongsTo*" for the new comment will be set to the parent post.

In case you have created the property "*author*" for your post, it will not work if the user of that id has not been loaded into the store. The code given below best describes this: **this.store.createRecord('post', {**

title: 'Ember is great',

body: 'Lorem ipsum',

```
 author: this.store.findRecord('user', 1) });
```

However, the relationship can easily be set once the promise has been fulfilled:

```
let p = this.store.createRecord('post', {
 title: 'Ember is Great',
 body: 'Lorem ipsum'
                                                    });
this.store.findRecord('user', 1).then(function(user) {
 post.set('author', user);
                                                    });
```

Updating of Existing Records

Sometimes, we might need to set relationships on records which are already in existence. This can simply be done by use of the "*belongsTo*" relationship. This is shown in the code given below: **let p = this.store.peekRecord('post', 1); let comm = this.store.peekRecord('comment', 1); comm.set('post', p);**

comm.save();

Alternatively, the record can be pushed into the relationship "*hasMany*" so as to update it. This is shown below: **let p = this.store.peekRecord('post', 1); let comm = this.store.peekRecord('comment', 1);**

p.get('comments').pushObject(comm);

p.save();

How to remove Relationships

For a "*belongsTo*" relationship to be removed, we can just set it to null. This will remove it from the side of "*hasMany*" relationship. Consider the code given below which best demonstrates this: **let comm = this.store.peekRecord('comment', 1); comm.set('post', null);**

comm.save();

A record can also be removed from a "*hasMany*" relationship. Consider the code given below which best shows how this can be done: **let p = this.store.peekRecord('post', 1); let comm = this.store.peekRecord('comment', 1);**

p.get('comments').removeObject(comm); p.save();

Relationships as Promises

Whenever you are working relationships, it is good for you to always remember that these will return promises. Suppose that we are to work on the asynchronous comments of a post, we should wait until our promise has been fulfilled. Consider the code given below: **let p = this.store.peekRecord('post', 1);**

p.get('comments').then((comments) => {

// we can now work with the comments });

The same will be applied to the relationship "*hasMany*" as shown below: **let comm = this.store.peekRecord('comment', 1); comm.get('post').then((p) => {**

// the post will be available here

});

Handlebars templates are updated automatically for reflection of a resolved promise. A list of comments can be displayed in a post as shown below: **<ul>**

{{#each p.comments as |comm|}}

**
{{comm.id}}
**

{{/each}}

**
**

Pushing Records into Store

A store can be thought of as a cache which contains all the records which have been loaded by the app.

For you to push records into a store, you just have to call the "*push()*" method. Consider a situation in which we are in need of preloading some data into the store once an app has been launched for the first time. This can easily be done using "*route:application.*" Consider the code given below: **export default DS.Model.extend({**

title: DS.attr(),

artist: DS.attr(),

songCount: DS.attr() });

Consider the next piece of code: **export default Ember.Route.extend({**

model() {

this.store.push({

data: [{

id: 1,

type: 'album', attributes: {

title: 'Very Few Moving Parts', artist: 'John Joel', songCount: 5

},

relationships: {}

}, {

id: 2,

type: 'album', attributes: {

title: 'Our God is Great all the time', artist: 'Steve John', songCount: 3

},

relationships: {}

}]

});

}

});

If there is a need for the data to be normalized by the default serializer of the model before it can be pushed to the store, one can use the method "*story.pushPayLoad().*"This is shown in the code given below: **export default DS.RestSerializer.extend({**

normalize(typeHash, hash) {

hash['songCount'] = hash['song_count']

delete hash['song_count']

```
  return this._super(typeHash, hash); }
})
```

The next code should be as shown below: **export default**

```
Ember.Route.extend({
 model() {
  this.store.pushPayload({
   albums: [
                                        {
     id: 1,
     title: 'Very Fever Moving Parts', artist: 'John Joel', song_count: 5
    },
                                        {
     id: 2,
     title: 'Our God is Great all the time', artist: 'Steve John', song_count:
2
                                        }
                                        ]
                                       });
                                        }
                                       ));
```

});

The method “*push()*” is very important when it comes to working with endpoints which are complex. Your application might have an endpoint which is tasked with the performing of some business logic and create several records. The code given below best demonstrates this: **export default Ember.Route.extend({**

```
 actions: {
   confirm: function(d) {
     $.ajax({
       data: d,
       method: 'POST', url: 'process-payment'
     }).then((digInventory) => {
       this.store.pushPayload(digInventory); this.transitionTo('Thanks'); });
                                                }
                                                }
                                               });
```

Handling MetaData Along with your records returned from the store, some form of metadata will need to be handled. Metadata is just a kind of data which goes along with some specific model or a type instead of a record.

Pagination is a common way that metadata can be used. Consider a blog with many posts which cannot be displayed at once. These can be queried as shown below: **let res = this.store.query('post', {**

limit: 10,

offset: 0

});

To get different pages of the data, the offset can be changed in increments of 10. The problem is knowing the number of pages of data that you are having.

For us to get different pages of data, the offset can be set in increments of 10. For the case of the server, it needs to return the total number of the records as a metadata piece.

Each serializer is expecting the metadata to be returned differently. Consider the code given below: **{**

"post": {

"id": 1,

"title": "Ember is easy to learn for beginners", "comments": ["1", "2"],

"links": {

"user": "*posts*ember"

},

// ...

},

"meta": {

"total": 100

}

}

Regardless of the serializer that is used, the metadata will be extracted from the purpose. To read it, you can make use of *".get('meta')."*

This can be applied on the result of the "*store.query()"* call. Here is the code:

store.query('post').then((res) => {

let metad = res.get('meta'); })

If we have a belongsTo relationship, this can be done as follows: **let p =**

store.peekRecord('post', 1); p.get('author').then((author) => {

let metad = author.get('meta'); });

On a hasMany relationship, this can be done as follows: **let p =**

store.peekRecord('post', 1); p.get('comments').then((comments) => {

let metad = comments.get('meta'); });

Customizations of Adapters In Ember data, the Adapter is tasked with the determination of how the data is persisted to a backend store, such as the URL format and headers of a RESTful API.

For an endpoint having some consistent rules, one can define a "*adapter:application.*"This will get a priority over a default adapter, but it will be superseded by adapters which are model specific. This is shown in the code given below: **export default DS.JSONAPIAdapter.extend({**

// Application specific overrides go here });

For a model having some specific rules about how it communicates with the backend compared to the others, a model specific adapter can be creating by execution of the command "ember generate adapter adapter-name." A good example is that when you run the command "*ember generate adapter post,*" the following file will be created: **export default DS.JSONAPIAdapter.extend({**

namespace: 'api/v1'

});

By default, the Ember data will come with multiple built-in adapters. If you are in need of creating your own custom adapter, use these as your starting point.

URL Conventions

The adapter “JSONAPIAdapter” is good when it comes to determination of the URLS which it uses to communicate depending on the type of the model. Consider the example given below of when a post is asked for by ID:

store.find('post', 1).then(function(p) {

});

Pluralization Customization Sometimes, you may need to facilitate the process of pluralization of model names whenever you are generating route URLs . The Ember reflector can be used for this purpose. This is a library which can be used for the purpose of alternating words from plural to singular form and vice versa. For uncountable or irregular pluralizations, one can make use of “Ember.Inflector.inflector.” This can easily be done as shown below: **// setting up the Ember.Inflector import './models/custom-inflector-rules';** The next code is given below: **import Inflector from 'ember-inflector'; const inflect = Inflector.inflector; inflect.irregular('formula', 'formulae'); inflect.uncountable('advice'); // Meeting the expectation of Ember Inspector of an export export default {};**

Customizing Endpoint Path The property "*namespace*" will be used for prefixing requests having a specific URL namespace. This is shown in the code given below: **export default DS.JSONAPIAdapter.extend({**

namespace: 'api/1'

});

That is how it can be done.

Host Customizations The adapter targets the current domain by default. If you are in need of specifying a new domain, you just have to set the property "*host*" on the new adapter. The code given below best shows how this can be done: **export default DS.JSONAPIAdapter.extend({**

host: 'https://api.sample.com'

});

That is it.

Path Customizations The method "*pathForType*" can be overridden as shown in the code given below: **export default DS.JSONAPIAdapter.extend({**

pathForType: function(type) {

return Ember.String.underscore(type); }

```
});
```

In the above case, we did not need to pluralize the names of the model, and we needed to use the underscore rather than the camelCase.

Headers Customizations Some APIs need HTTP headers, and a good example is when there is a need for provision of an API key. Arbitrary headers can then be set as the key/value pairs on the header object of the JSONAPIAdapter, and the Ember data will send along each of our Ajax requests. Consider the code given below: **export default DS.JSONAPIAdapter.extend({**

headers: {

'API_KEY': 'secret key', 'ANOTHER_HEADER': 'A header value'

}

});

One can also use *"headers"* as the computed property for the purpose of supporting dynamic headers. Consider the example given below in which the headers will be generated by use of computed property depending on the session service. Here is the code for the example: **export default DS.JSONAPIAdapter.extend({**

session: Ember.inject.service('session'), headers: Ember.computed('session.authToken', function() {

return {

'API_KEY': this.get('session.authToken'), 'ANOTHER_HEADER': 'A

header value'

};

})

});

In some cases, the dynamic headers will need some data from the object outside the observer system of the Ember. The volatile function can also be used for the purpose of setting the property into a non-cached mode which will cause re-computation of the headers with each request. This is shown in the code given below: **export default DS.JSONAPIAdapter.extend({**

headers: Ember.computed(function() {

return {

'API_KEY': Ember.get(document.cookie.match(/apiKey\=([^;]*)/), '1'), 'ANOTHER_HEADER': 'A header value'

};

}).volatile()

});

Authoring Adapters

The property “*defaultSerializer”* will be used for the purpose of specifying the serializer which will be used by the adapter. This is only used when a “serializer:application” or a model specific adapter has not been specified. Specification of the “serializer:application” is very easy in any application. Consider the example given below demonstrating how this can be done: **export default DS.JSONAPIAdapter.extend({**

defaultSerializer: '-default'

});

Customizing Serializers In Ember data, serializers are used for formatting data which is sent to and received from our backend store. The default setting is that the Ember data will use the JSON API format for serialization of data.

JSON API Document Consider the response given below for our request: **{**

"data": {

"type": "people", "id": "567", "attributes": {

"first-name": "John", "last-name": "Joel"

}

}

}

In the above example, we have a response from a request to "*people567.*" A response having multiple records can have an array in the data property as shown below: **{**

"data": [{

"type": "people", "id": "567", "attributes": {

"first-name": "John", "last-name": "Joel"

}

}

}, {

"type": "people", "id": "345", "attributes": {

"first-name": "Mercy", "last-name": "Donald"

}

}]

}

Sideloaded Data This is a type of data which is not part of a primary request but has some linked relationships which should be added in an array under the key "*included.*" Consider the code given below: **{**

"data": {

"type": "blogs",

"id": "1",

"attributes": {

"title": "I really like and enjoy blogging!"

},

"links": {

"self": "http://mysite.com*blogs*1"

},

"relationships": {

"comments": {

"data": [

{ "type": "comments", "id": "5" }, { "type": "comments", "id": "12"

}

]

}

}

```
},
"included": [{
  "type": "comments",
  "id": "5",
  "attributes": {
    "body": "First!"
  },
  "links": {
    "self": "http://mysite.com/comments/5"
                                                    }
                                                  }, {
  "type": "comments",
  "id": "12",
  "attributes": {
    "body": "Ember is very easy to learn"
  },
  "links": {
    "self": "http://mysite.com/comments/12"
                                                    }
                                                  }]
                                                   }
```

In the example given above, we have requested *"blogs1"* and the user also gave us the comments which are associated with the person. The above code shows the response that we get.

Customizing Serializers

In Ember data, “JSONAPISerializer” is used by default, but a custom serializer can be used for the purpose of overriding this. A custom serializer can be defined in two main ways. One can choose to define a custom serializer for the entire application by definition of the “application” serializer. This is shown in the code given below: **import DS from 'ember-data'; export default DS.JSONSerializer.extend({});** A serializer can also be defined for a specific model. A good example is when you have a “*post*” model, it is possible for you to also define a “*post*” serializer. This is shown in the code given below: **import DS from 'ember-data'; export default DS.JSONSerializer.extend({});** For the format of the data sent to the backend store to be changed, one can use the “*serialize()*” hook. This is shown in the code given below: Consider a situation in which the Ember data gives us the following JSON API response: **{**

```
 "data": {
   "attributes": {
     "id": "1",
     "name": "The Product", "amount": 200,
     "currency": "USD"
   },
   "type": "product"
```

}

}

However, our server might be expecting to get data in this format: **{**

"data": {

"attributes": {

"id": "1",

"name": "The Product", "cost": {

"amount": 200,

"currency": "USD"

}

},

"type": "product"

}

}

The data can then be changed as shown below: **import DS from 'ember-data';**

export default DS.JSONSerializer.extend({

serialize(snapshot, opts) {

var json = this._super(...arguments); json.data.attributes.cost = {

amount: json.data.attributes.amount, currency:

json.data.attributes.currency };

delete json.data.attributes.amount; delete json.data.attributes.currency;

return json;

},

});

In case the backend store is providing data in any format other than the JSON API, one can use the hook "*normalizeResponse().*" By use of the example given above, the server will provide us with data which looks as follows: **{**

"data": {

"attributes": {

"id": "1",

"name": "A Product",

"cost": {

"amount": 200,

"currency": "USD"

}

},

"type": "product"

}

}

The above should be changed to the following: **{**

"data": {

"attributes": {

"id": "1",

"name": "A Product",

"amount": 200,

"currency": "USD"

},

"type": "product"

}

}

It should then be done as shown below: **import DS from 'ember-data'; export**

default DS.JSONSerializer.extend({

normalizeResponse(store, primModelClass, payload, id, reqType) {
payload.data.attributes.amount = payload.data.attributes.cost.amount;
payload.data.attributes.currency = payload.data.attributes.cost.currency;
delete payload.data.attributes.cost; return this._super(...arguments); },

});

IDs

If you are in need of tracking the unique records contained in a store, the Ember data expects that each record should have a unique id property in the payload. For each record of a specific type, the id should also be unique. If the backend had used a different key rather than the "*id,*" one can use the primary key property of the serializer so as to transform the id properly during the process of serialization and deserialization of the data. This is shown in the code given below: **export default DS.JSONSerializer.extend({**

primaryKey: '_id'

});

Attribute Names

The convention in Ember data is to camelize the attribute names on a particular model. Consider the example given below: **export default DS.Model.extend({**

fName: DS.attr('string'),

lName: DS.attr('string'), isPersonOfTheYear: DS.attr('boolean') });

The “JSONAPISerializer” is expecting the attributes to be dasherized in a document payload returned by the server. This is shown in the code given below:

{

"data": {

"id": "44",

"type": "people",

"attributes": {

"first-name": "John",

"last-name": "Joel",

"is-person-of-the-year": true }

}

}

In case the attributes returned by the server use a very different convention, one can use the method "*keyForAttribute()*" of the serializer for conversion of the attribute name in the model to a key in the JSON payload. Consider the code given below: **export default DS.JSONAPISerializer.extend({**

keyForAttribute: function(attribute) {

return Ember.String.underscore(attribute); }

});

A custom serializer can be used for mapping irregular keys. The object "*attrs*" can be used for the purpose of declaring a simple mapping between different property names in the DS. Consider the sample code given below: **export default DS.Model.extend({**

lName: DS.attr('string')

});

What we have done is that we have created a custom serializer for our model and then overrode the property "*attrs.*" The next piece of code should be as follows: **export default DS.JSONAPISerializer.extend({**

attrs: {

lName: 'lastNameOfPerson'

}

```
}
});
```

Relationships The ID property should be used for referencing to some other records. Consider a situation in which your model has the hasMany relationship shown below: **export default DS.Model.extend({**

comments: DS.hasMany('comment', { async: true }) });

The JSON should then encode the relationship just as an array of types and IDs. This is shown in the code given below: **{**

"data": {

"type": "posts",

"id": "1",

"relationships": {

"comments": {

"data": [

{ "type": "comments", "id": "5" }, { "type": "comments", "id": "12"

}

]

}

}

}

}

Consider a situation in which you have a model which is as follows: **export default DS.Model.extend({**

origPost: DS.belongsTo('post') });

The relationship should then be encoded by JSON into another record as an ID. This is shown in the code given below: **{**

"data": {

"type": "comment",

"id": "1",

"relationships": {

"original-post": {

"data": { "type": "post", "id": "5" }, }

}

}

}

If these naming conventions are needed, then they can be overridden by use of the method "*keyForRelationship.*" Consider the code given below which best demonstrates this: **export default DS.JSONAPISerializer.extend({**

keyForRelationship: function(key, relship) {

return key + 'Ids'; }

});

Creation of Custom Transformations Under certain circumstances, built-in types such as the string, Boolean, and others may not be adequate for us to implement what we need. A good example of this is when a particular server gives you a non-standard date format.

In Ember data, we may have new JSON transforms which have been registered to be used as attributes. The code given below best describes this: **export default DS.Transform.extend({**

serialize: function(val) {

return [value.get('a'), value.get('b')]; },

deserialize: function(val) {

return Ember.Object.create({ a: value[0], b: value[1] }); }

});

The next code should then be as follows: **export default DS.Model.extend({**

position: DS.attr('coordinate-point') });

After receiving "*coordinatePoint"* from the API, it is expected to be an array as shown below: **{**

cursor: {

position: [5,9]

}

}

Once it has been loaded on an instance of a model, it will act as an object as shown below: **var c = store.findRecord('cursor', 1); c.get('position.x'); //=> 5 c.get('position.y'); //=> 9**

Once the position has been modified and then saved, it will be passed through the function "*serialize"* in our transform and then be presented like an array in JSON.

JSONSerializer

For this to be used in an app, one has to define the “serializer:application” which is just an extension of JSONSerializer. Consider the code given below which shows how this is done: **export default DS.JSONSerializer.extend({**

// ...

});

In the case of some requests, the expected result should have one record. It is expected that the response should be a JSON response with a format similar to the one given below: **{**

"id": "1",

"title": "Ember is easy to Learn", "tag": "rails",

"comments": ["1", "2"]

}

For results expected to give 0 or more records, it is expected that the result should be a JSON array which looks as follows: **[{**

"id": "1",

"title": "Ember is easy to Learn", "tag": "ember",

"comments": ["1", "2"]

}, {

"id": "2",

"title": "When I was young, I aspired to become an athlete!", "tag":

"w3c",

"comments": ["3"]

}]

EmbeddedRecordMixin

In Ember data, we are encouraged to sideload our relationships, but sometimes, when one is working with legacy APIs, one may discover that they need to deal with JSON which contains relationships which have been embedded inside other records. The “EmbeddedRecordsMixin” was created for the purpose of solving this problem.

To setup the embedded records, just include the mixin during the process of extension of a serializer and then define and perform a configuration of the embedded relationships.

Consider a situation in which the model “*post*” has a “*author*” record which looks as follows: **{**

"id": "1",

"title": "Ember is easy to Learn", "tag": "ember",

"authors": [

{

"id": "2", "name": "Mark"

}

]

}

The relationship would then be defined as shown below: **export default DS.JSONSerializer.extend(DS.EmbeddedRecordsMixin, {**

attrs: {

authors: {

serialize: 'records', deserialize: 'records'

}

}

});

In case you have defined yourself and you need to serialize and deserialize the embedded relationship, the shorthand option “{ embedded: 'always' }” can be used. Consider the following example which is similar to the one given below: **export default DS.JSONSerializer.extend(DS.EmbeddedRecordsMixin, {**

attrs: {

authors: { embedded: 'always' }

}

});

Consider a situation in which you want to read an embedded record during the process of extraction of a JSON payload, but you only need to include the id of the relationship whenever you are serializing a record. This can easily be done by use of the option “serialize: 'ids.'” To opt out of serialization of the relationship, you can use the option “serialize: false.” The code given below demonstrates how this can be done: **export default DS.JSONSerializer.extend(DS.EmbeddedRecordsMixin, {**

```
 attrs: {
   author: {
     serialize: false, deserialize: 'records'
   },
   comments: {
     deserialize: 'records', serialize: 'ids'
                                                    }
                                                    }
                                                   });
```

EmbeddedRecordsMixin Defaults

If you do not need to overwrite the property “*attrs”* for a particular relationship, then the following will be the behavior of the “EmbeddedRecordsMixin”:

BelongsTo: { serialize: 'id', deserialize: 'id' }

HasMany: { serialize: false, deserialize: 'ids' }

Authoring Serializers For those who need to create a custom serializer, we are encouraged to start with a JSONAPISerializer or a JSONSerializer, and then extend one of these so as to match our needs. However, in case the payload is different from one of the serializers, consider a situation in which you are given the model "*posts*" which is given below: **export default DS.Model.extend({**

title: DS.attr('string'), tag: DS.attr('string'), comments: hasMany('comment', { async: false }), relatedPosts: hasMany('blogs') });

The "*store.push*" will then accept an object which is as follows: **{**

```
data: {
  id: "1",
  type: 'post',
  attributes: {
    title: "Ember is easy to Learn", tag: "ember",
  },
  relationships: {
    comments: {
      data: [{ id: "1", type: 'comment' }, { id: "2", type: 'comment' }], },
    relatedPosts: {
```

data: {

related: "*api*v1/blogs/1/related-posts/"

}

}

}

}

For each serialized record to be convertible into an Ember data record, the above format must be followed.

Chapter 3- Application Concerns

Dependency Injection

This technique is used in EmberJS for declaration and instantiation of a class of objects and the dependencies between them. Applications and the instances of applications will serve a role in the DI implementation of Ember.

Factory Registrations

A factory can be used for representation of part of an app, such as a template, a route, or a custom class. A good example of this is the index template which is registered with the key “template:index.”

Note that factory registrations have to be done inside an application or in application instance initializers, but the former way is much more common.

Consider the example given below, which shows how an application initialize can register a “*logger”* factory using the key “logger:main”: **export function initialize(app) {**

var Logger = Ember.Object.extend({

log(n) {

console.log(n); }

});

app.register('logger:main', Logger); }

export default {

name: 'logger',

initialize: initialize };

Registering Instantiated Objects The default setting is that Ember attempts to instantiate a registered factory after looking up. During the process of registration of instantiated object of a particular class, you have to use the option “instantiate: false” so that you can avoid it being re-instantiated during look up.

In the example given below, the logger is an object in JavaScript which should be returned as it is during look up. Here is the code for the example: **export function initialize(app) {**

var logger = {

log(n) {

console.log(n); }

};

app.register('logger:main', logger, { instantiate: false }); }

export default {

name: 'logger',

initialize: initialize };

Singletons vs. Non-Singletons Registration

In Ember data, registrations are seen as singletons. This means that an instance has to be created after looking up, and the same instance will be cached and returned from the following lookups.

In the example given below, the class "*message*" will be registered as a non-singleton: **export function initialize(app) {**

var Msge = Ember.Object.extend({

text: ''

});

app.register('notification:message', Message, { singleton: false }); }

export default {

name: 'notification', initialize: initialize };

Factory Injections Once the registration of the factory has been done, we can inject it when we need. Injection of factories can be done in whole types of the factories having type injections. Consider the example given below: **export function initialize(app) {**

```
 var Logger = Ember.Object.extend({
   log(n) {
     console.log(n);
                                                    }
                                                  });
 app.register('logger:main', Logger); app.inject('route', 'logger',
'logger:main'); }
export default {
 name: 'logger',
 initialize: initialize };
```

Due to this kind of type injection, the factories of type "*route*" will be instantiated with the logger which has been injected. The logger value will be obtained from the factory with the name "logger:main."

Consider the example given below which shows how the app can be accessed by

the injected logger. Here is the example: **export default Ember.Route.extend({**

activate() {

// The property logger is injected into all the routes

this.get('logger').log('Entered your index route!'); }

});

When using a full key, the injections can be made on a full key as shown below:

app.inject('route:index', 'logger', 'logger:main');

Ad Hoc Injections Dependency injections are declared directly on the Ember classes by the use of “Ember.inject.” Currently, this property is in support of injection of services and controllers.

Consider the code given below which shows how the service “*shopping-cart”* can be injected on the component “cart-contents” as the “*cart”* property. Here is the code: **export default Ember.Component.extend({**

cart: Ember.inject.service('shopping-cart') });

If you are in need of injecting a service with similar name as the property, just leave off the name of the service. The code given below demonstrates this:

export default Ember.Component.extend({

shoppingCart: Ember.inject.service() });

Instance initializers usually receive an instance of an application as the argument, providing us with an opportunity to lookup the instance of the registered factory. Consider the code given below: **export function initialize(appInstance) {**

let logger = appInstance.lookup('logger:main'); logger.log('Hello from an instance initializer!'); }

export default {

name: 'logger',

initialize: initialize

};

It is also possible for us to get an instance of an application from a factory set. Consider the example given below which plays songs having different audio services depending on the "*audioType"* of the song. Here is the code: **import Ember from 'ember'; const {**

Component,

computed,

getOwner

} = Ember;

// Usage:

//

// {{play-audio song=song}}

//

export default Component.extend({

audioService: computed('song.audioType', function() {

let appInstance = getOwner(this); let audioType = this.get('song.audioType'); return appInstance.lookup(`service:audio-

```
${audioType}`); }),

 click() {
   let p = this.get('audioService'); p.play(this.get('mysong.file')); }

                                                    });
```

Initializers These provide us with an opportunity to configure the app during the boot process. There exists two types of initializers, application instance initializers and application initializers.

The application initializers are executed during the boot process, and they provide us with the method that dependency injections can be configured in an app. The other type of initializers are configured during the loading of an instance of an application. They provide us with a way to configure the initial state of our app, and setting up of application instances which are local to the instance of our app.

Application Initializers Application initializers are created using the initializer generator of the Ember CLI.

The command given below best describes this: **ember generate initializer shopping-cart** We can then customize the initializer "*shopping-cart"* so as to inject the property "*cart"* in all the routes contained in our application. The code given below best describes this: **export function initialize(app) {**

app.inject('route', 'cart', 'service:shopping-cart'); };

export default {

name: 'shopping-cart',

initialize: initialize

};

Application Instance Initializers These can be created using the generator "instance-initializer" of the Ember CLI. This is shown in the code given below: **ember generate instance-initializer logger** Consider the code given below which shows how some simple logging can be implemented so as to indicate that the instance has already been booted. This is shown in the code given below: **export function initialize(appInstance) {**

var logger = appInstance.lookup('logger:main'); logger.log('Hi from our

instance initializer!'); }

export default {

name: 'logger',

initialize: initialize

};

Specifying Initializer Order If you need to control the order in which initializers run, the "*before*" and "*after*" options can be used for this purpose. Consider the code given below: **export function initialize(app) {**

// ...add your code here ...

};

export default {

name: 'configReader', before: 'websocketInit', initialize: initialize };

The next code should then be as follows: **export function**

initialize(application) {

// ... add your code here...

};

export default {

name: 'websocketInit', after: 'configReader', initialize: initialize };

Services

The property "*Ember.service"* is an object in Ember which can be used and made available at various parts of one's app. The following are the areas in which the services are used:

- Geolocation.
- User/session authentication.
- Web Sockets.
- Server-backed API calls which cannot fit Ember Data.
- Third-party APIs.
- Server-sent events or the notifications.Logging.

Definition of Services The “service” generator in Ember CLI can be used for the purpose of generating services. Consider the following command which can be used for the purpose of creating a “*shoppingCart”* service: **ember generate service shopping-cart** The services have to extend the base class “*Ember.Service.”* This is shown in the code given below: **export default Ember.Service.extend({**

});

Like the objects in Ember, a service has to be initialized, and this should have methods and properties of its own. Consider the shopping cart service shown below which best demonstrates how this works: **export default Ember.Service.extend({**

```
 items: null,
 init() {
   this._super(args); this.set('items', []); },
 add(item) {
   this.get('items').pushObject(item); },
 remove(item) {
   this.get('items').removeObject(item); },
 empty() {
```

this.get('items').setObjects([]); }

});

Accessing Services For a service to be accessed, one can choose to inject it into any object which is container-resolved, such as another service or a component by use of the function "*Ember.inject.service.*" There are two ways that a function can be used. One can choose to invoke it without arguments, or pass the registered name of your service to it.

In case no arguments are passed, then the services have to be loaded based on the name of its variable key. To load the shopping cart service with no arguments, do it as shown below: **export default Ember.Component.extend({**

//ant to load the service in the file *app*services/shopping-cart.js

shoppingCart: Ember.inject.service() });

The other method involves using the name of the service as the argument. This is shown in the code given below: **export default Ember.Component.extend({**

//want to load our service in the file *app*services/shopping-cart.js cart: Ember.inject.service('shopping-cart') });

With the above code, the shopping cart service will be injected into our component and then be made available to the cart property. Properties which have been injected are lazy loaded, meaning that instantiation of the service will not be done until the property has not been called explicitly. This means that the

function “*get”* has to be used for accessing the services of a particular component, otherwise, you will get an undefined error.

After loading, the service persists until the service exits.

export default Ember.Component.extend({

Consider the code given below: **cart: Ember.inject.service('shopping-cart'), actions: {**

remove(item) {

this.get('cart').remove(item); }

}

});

In the above code, we have added the remove action to the component “cart-contacts.” Once it has been injected into a component, the service will also be used in a template. Note the cart which has been used in the code given below for getting the data from the cart: **<ul>**

{{#each cart.items as |item|}}

<li>

```
    {{item.name}}
    <button {{action "remove" item}}>Remove</button> </li>
 {{/each}}
</ul>
```

The Run Loop Most of the internals and the code we write in Ember for our apps occur in a run loop. The run loop is used for the purpose of batching and ordering the work in a way which is more efficient and effective.

We get a huge benefit when we batch similar works. For web browsers to do something which is related, they have to batch the changes made to the DOM. Consider the HTML snippet given below: ** **

** **

** **

You can then run the following code: **foo.style.height = '500px' // write foo.offsetHeight //write**

bar.style.height = '400px' // write bar.offsetHeight // read

baz.style.height = '200px' // write baz.offsetHeight // read

In the above example, the sequence of the code has forced the browser to recalculate the style and the layout after each step. However, if it was possible for us to batch the similar jobs together, the browser would only be needed to recalculate the style and the layout only once. This is shown in the code given below: **foo.style.height = '500px' // write bar.style.height = '400px' // write baz.style.height = '200px' // write foo.offsetHeight // read**

bar.offsetHeight // read

baz.offsetHeight // read

The good thing about this is that it applies to many other types of work. When we batch similar works, we find it easy for us to pipeline and further the optimization.

Consider the example given below, which has been optimized in Ember: **var MyUser = Ember.Object.extend({**

fName: null,

lName: null,

fullName: Ember.computed('fName', 'lName', function() {

return `${this.get('fName')} ${this.get('lName')}`; })

});

The template for displaying the attributes is as shown below: **{{fName}}**

{{fullName}}

The following code can be executed without using a run loop as shown below:

var myuser = User.create({ fName: 'John', lastName: 'Boss' });

myuser.set('fName', 'Joel');

// {{fName}} and {{fullName}} are then updated myuser.set('lName', 'Donald');

// {{lName}} and {{fullName}} are updated With the above, your browser should re-render the template twice.

Note that if the loop in the above code has been run, the browser will only re-render the template once all of the attributes have been set. Consider the code given below: **var myuser = User.create({ fName: 'John', lastName: 'Boss' });**

myuser.set('fName', 'Joel');

user.set('lName', 'Donald');

user.set('fName', 'John ');

user.set('lastName', 'Boss');

In the above example, we have a run loop, since the attributes of the user will end up at the same values as before the execution, and the template will not be re-rendered.

A run loop should be begun once a callback has been fired. Consider the example given below: **$('a').click(() => {**

Ember.run(() => { // beginning the loop // Code which results in the jobs being scheduled should go here }); // ending the loop, jobs are flushed and

then executed });

That is the new syntax which we currently have for functions.

A non-Ember async callbacks should be wrapped in “Ember.run.” if you don’t do this, Ember will approximate and then define a beginning and an end for you. Consider the callback given below: **$('a').click(() => {**

console.log('Doing the things...'); Ember.run.schedule('actions', () => {

// Add more things

});

});

That is it for now.

Conclusion

We have come to the conclusion of this guide. Ember is one of the available JavaScript frameworks in use today. This framework is mostly used when we are in need of developing the client side of our web applications. For it to structure the apps, it relies on the MVC architecture.

This framework has a number of features which we can take advantage of so as to enhance our apps. In Ember JS, controllers are of great importance, and most people confuse them with components. Very soon, it is believed that the controllers will be replaced with components. Controllers are used for the purpose of controlling state, and this is determined by the kind of route which is currently being used. A controller is also used when passing information from a component to a route, as this has to pass through it. Dependencies need to be established between controllers.

This is usually the case when we are in need of nesting resources. We have to come up with a mechanism as to how to establish a connection between the controllers. It is good for you to know how to create custom transformations in Ember JS. This is because the data types which might be available may not be

sufficient for you. There are custom transformations in Ember JS which can assist you in this. My hope is that this book has helped you understand how to create apps in Ember JS. Enjoy!