Single Page ToDo Application With Backbone.js

by Krasimir Tsonev 13 Jun 2014

Difficulty: Beginner Length: Medium Languages: English

Web Development | Front-End | JavaScript | jQuery | BackboneJS | Single Page Web Apps

Backbone.js is a JavaScript framework for building flexible web applications. It comes with Models, Collections, Views, Events, Router and a few other great features. In this article we will develop a simple ToDo application which supports adding, editing, and removing tasks. We should also be able to mark a task as *done* and archive it. In order to keep this post's length reasonable, we will not include any communication with a database. All the data will be kept on the client-side.

Setup

Here is the file structure which we'll use:

```
css
    └─── styles.css
js
    └─── collections
        └─── ToDos.js
    └─── models
        └─── ToDo.js
    └─── vendor
       └─── backbone.js
       └─── jquery-1.10.2.min.js
       └─── underscore.js
    └─── views
    └─── App.js
  └─── index.html
```

There are few things which are obvious, like `/css/styles.css` and `/index.html`. They contain the CSS styles and the HTML markup. In the context of Backbone.js, the model is a place where we keep our data. So, our ToDos will simply be models. And because we will have more than one task, we will organize them into a collection. The business logic is distributed between the views and the main application's file, `App.js`. Backbone.js has only one hard dependency - Underscore.js. The framework also plays very well with jQuery, so they both go to the `vendor` directory. All we need now is just a little HTML markup and we are ready to go.

```
<!doctype html>
<html>
  <head>
        <title>My TODOs</title>
        <link rel="stylesheet" type="text/css" href="css/styles.css" />
    </head>
    <body>
        <div class="container">
            <div id="menu" class="menu cf"></div>
            <h1></h1>
            <div id="content"></div>
        </div>
        <script src="js/vendor/jquery-1.10.2.min.js"></script>
        <script src="js/vendor/underscore.js"></script>
        <script src="js/vendor/backbone.js"></script>
        <script src="js/App.js"></script>
        <script src="js/models/ToDo.js"></script>
        <script src="js/collections/ToDos.js"></script>
        <script>
            window.onload = function() {
                // bootstrap
            }
        </script>
    </body>
</html>
```

As you can see, we are including all the external JavaScript files towards the bottom, as it's a good practice to do this at the end of the body tag. We are also preparing the bootstrapping of the application. There is container for the content, a menu and a title. The main navigation is a static element and we are not going to change it. We will replace the content of the title and the `div` below it.

Planning the Application

It's always good to have a plan before we start working on something. Backbone.js doesn't have a super strict architecture, which we have to follow. That's one of the benefits of the framework. So, before we start with the implementation of the business logic, let's talk about the basis.

Namespacing

A good practice is to put your code into its own scope. Registering global variables or functions is not a good idea. What we will create is one model, one collection, a router and few Backbone.js views. All these elements should live in a private space. `App.js` will contain the class which holds everything.

```
// App.js
var app = (function() {

    var api = {
        views: {},
        models: {},
        collections: {},
        content: null,
        router: null,
        todos: null,
        init: function() {
            this.content = $("#content");
        },
        changeContent: function(el) {
            this.content.empty().append(el);
            return this;
        },
        title: function(str) {
            $("h1").text(str);
            return this;
        }
    };
    var ViewsFactory = {};
    var Router = Backbone.Router.extend({});
    api.router = new Router();

    return api;

})();
```

Above is a typical implementation of the revealing module pattern. The `api` variable is the object which is returned and represents the public methods of the class. The `views`, `models` and `collections` properties will act as holders for the classes returned by Backbone.js. The `content` is a jQuery element pointing to the main user's interface container. There are two helper methods here. The first one updates that container. The second one sets the page's title. Then we defined a module called `ViewsFactory`. It will deliver our views and at the end, we created the router.

You may ask, why do we need a factory for the views? Well, there are some common patterns while working with Backbone.js. One of them is related to the creation and usage of the views.

```
var ViewClass = Backbone.View.extend({ /* logic here */ });
var view = new ViewClass();
```

It's good to initialize the views only once and leave them alive. Once the data is changed, we normally call methods of the view and update the content of its `el` object. The other very popular approach, is to recreate the whole view or replace the whole DOM element. However, that's not really good from a performance point of view. So, we normally end up with a utility class which creates one instance of the view and returns it when we need it.

Components Definition

We have a namespace, so now we can start creating components. Here is how the main menu looks:

```
// views/menu.js
app.views.menu = Backbone.View.extend({
    initialize: function() {},
    render: function() {}
});
```

We created a property called `menu` which holds the class of the navigation. Later, we may add a method in the factory module which creates an instance of it.

```
var ViewsFactory = {
    menu: function() {
        if(!this.menuView) {
            this.menuView = new api.views.menu({
                el: $("#menu")
            });
        }
        return this.menuView;
    }
};
```

Above is how we will handle all of the views, and it will ensure that we get only one and of the same instance. This technique works well, in most cases.

Flow

The entry point of the app is `App.js` and its `init` method. This is what we will call in the `onload` handler of the `window` object.

```
window.onload = function() {
    app.init();
}
```

After that, the defined router takes control. Based on the URL, it decides which handler to execute. In Backbone.js, we don't have the usual Model-View-Controller architecture. The Controller is missing and most of the logic is put into the views. So instead, we wire the models directly to methods, inside the views and get an instant update of the user interface, once the data has changed.

Managing the Data

The most important thing in our small project is the data. Our tasks are what we should manage, so let's start from there. Here is our model definition.

```
// models/ToDo.js
app.models.ToDo = Backbone.Model.extend({
    defaults: {
        title: "ToDo",
        archived: false,
        done: false
    }
});
```

Just three fields. The first one contains the text of the task and the other two are flags which define the status of the record.

Every thing inside the framework is actually an event dispatcher. And because the model is changed with setters, the framework knows when the data is updated and can notify the rest of the system for that. Once you bind something to these notifications, your application will react on the changes in the model. This is a really powerful feature in Backbone.js.

As I said in the beginning, we will have many records and we will organize them into a collection called `ToDos`.

```
// collections/ToDos.js
app.collections.ToDos = Backbone.Collection.extend({
    initialize: function(){
        this.add({ title: "Learn JavaScript basics" });
        this.add({ title: "Go to backbonejs.org" });
        this.add({ title: "Develop a Backbone application" });
    },
    model: app.models.ToDo
    up: function(index) {
        if(index > 0) {
            var tmp = this.models[index-1];
            this.models[index-1] = this.models[index];
            this.models[index] = tmp;
            this.trigger("change");
        }
    },
    down: function(index) {
        if(index < this.models.length-1) {
            var tmp = this.models[index+1];
            this.models[index+1] = this.models[index];
            this.models[index] = tmp;
            this.trigger("change");
        }
    },
    archive: function(archived, index) {
        this.models[index].set("archived", archived);
    },
    changeStatus: function(done, index) {
        this.models[index].set("done", done);
    }
});
```

The `initialize` method is the entry point of the collection. In our case, we added a few tasks by default. Of course in the real world, the information will come from a database or somewhere else. But to keep you focused, we will do that manually. The other thing which is typical for collections, is setting the `model` property. It tells the class what kind of data is being stored. The rest of the methods implement custom logic, related to the features in our application. `up` and `down` functions change the order of the ToDos. To simplify things, we will identify every ToDo with just an index in the collection's array. This means that if we want to fetch one specific record, we should point to its index. So, the ordering is just switching the elements in an array. As you may guess from the code above, `this.models` is the array which we are talking about. `archive` and `changeStatus` set properties of the given element. We put these methods here, because the views will have access to the `ToDos` collection and not to the tasks directly.

Additionally, we don't need to create any models from the `app.models.ToDo` class, but we do need to create an instance from the `app.collections.ToDos` collection.

```
// App.js
init: function() {
    this.content = $("#content");
    this.todos = new api.collections.ToDos();
    return this;
}
```

Showing Our First View (Main Navigation)

The first thing which we have to show, is the main application's navigation.

```
// views/menu.js
app.views.menu = Backbone.View.extend({
    template: _.template($("#tpl-menu").html()),
    initialize: function() {
        this.render();
    },
    render: function(){
        this.$el.html(this.template({}));
    }
});
```

It's only nine lines of code, but lots of cool things are happening here. The first one is setting a template. If you remember, we added Underscore.js to our app? We are going to use its templating engine, because it works good and it is simple enough to use.

```
_.template(templateString, [data], [settings])
```

What you have at the end, is a function which accepts an object holding your information in key-value pairs and the `templateString` is HTML markup. Ok, so it accepts an HTML string, but what is `$("#tpl-menu").html()` doing there? When we are developing a small single page application, we normally put the templates directly into the page like this:

```
// index.html
<script type="text/template" id="tpl-menu">
    <ul>
        <li><a href="#">List</a></li>
        <li><a href="#archive">Archive</a></li>
        <li class="right"><a href="#new">+</a></li>
    </ul>
</script>
```

And because it's a script tag, it is not shown to the user. From another point of view, it is a valid DOM node so we could get its content with jQuery. So, the short snippet above just takes the content of that script tag.

The `render` method is really important in Backbone.js. That's the function which displays the data. Normally you bind the events fired by the models directly to that method. However, for the main menu, we don't need such behavior.

```
this.$el.html(this.template({}));
```

`this.$el` is an object created by the framework and every view has it by default (there is a `$` infront of `el` because we have jQuery included). And by default, it is an empty `<div></div>`. Of course you may change that by using the `tagName` property. But what is more important here, is that we are not assigning a value to that object directly. We are not changing it, we are changing only its content. There is a big difference between the line above and this next one:

```
this.$el = $(this.template({}));
```

The point is, that if you want to see the changes in the browser you should call the render method before, to append the view to the DOM. Otherwise only the empty div will be attached. There is also another scenario where you have nested views. And because you are changing the property directly, the parent component is not updated. The bound events may also be broken and you need to attach the listeners again. So, you really should only change the content of `this.$el` and not the property's value.

The view is now ready and we need to initialize it. Let's add it to our factory module:

```
// App.js
var ViewsFactory = {
    menu: function() {
        if(!this.menuView) {
            this.menuView = new api.views.menu({
                el: $("#menu")
            });
        }
        return this.menuView;
    }
};
```

At the end simply call the `menu` method in the bootstrapping area:

```
// App.js
init: function() {
    this.content = $("#content");
    this.todos = new api.collections.ToDos();
    ViewsFactory.menu();
    return this;
}
```

Notice that while we are creating a new instance from the navigation's class, we are passing an already existing DOM element `$("#menu")`. So, the `this.$el` property inside the view is actually pointing to `$("#menu")`.

Adding Routes

Backbone.js supports the *push state* operations. In other words, you may manipulate the current browser's URL and travel between pages. However, we'll stick with the good old hash type URLs, for example `/#edit/3`.

```
// App.js
var Router = Backbone.Router.extend({
    routes: {
        "archive": "archive",
        "new": "newToDo",
        "edit/:index": "editToDo",
        "delete/:index": "delteToDo",
        "": "list"
    },
    list: function(archive) {},
    archive: function() {},
    newToDo: function() {},
    editToDo: function(index) {},
    delteToDo: function(index) {}
});
```

Above is our router. There are five routes defined in a hash object. The key is what you will type in the browser's address bar and the value is the function which will be called. Notice that there is `:index` on two of the routes. That's the syntax which you need to use if you want to support dynamic URLs. In our case, if you type `#edit/3` the `editToDo` will be executed with parameter `index=3`. The last row contains an empty string which means that it handles the home page of our application.

Showing a List of All the Tasks

So far what we've built is the main view for our project. It will retrieve the data from the collection and print it out on the screen. We could use the same view for two things - displaying all the active ToDos and showing those which are archived.

Before to continue with the list view implementation, let's see how it is actually initialized.

```
// in App.js views factory
```

```
list: function() {
    if(!this.listView) {
        this.listView = new api.views.list({
            model: api.todos
        });
    }
    return this.listView;
}
```

Notice that we are passing in the collection. That's important because we will later use `this.model` to access the stored data. The factory returns our list view, but the router is the guy who has to add it to the page.

```
// in App.js's router
list: function(archive) {
    var view = ViewsFactory.list();
    api
    .title(archive ? "Archive:" : "Your ToDos:")
    .changeContent(view.$el);
    view.setMode(archive ? "archive" : null).render();
}
```

For now, the method `list` in the router is called without any parameters. So the view is not in `archive` mode, it will show only the active ToDos.

```
// views/list.js
app.views.list = Backbone.View.extend({
    mode: null,
    events: {},
    initialize: function() {
        var handler = _.bind(this.render, this);
        this.model.bind('change', handler);
        this.model.bind('add', handler);
        this.model.bind('remove', handler);
    },
    render: function() {},
    priorityUp: function(e) {},
    priorityDown: function(e) {},
    archive: function(e) {},
    changeStatus: function(e) {},
    setMode: function(mode) {
        this.mode = mode;
        return this;
    }
});
```

The `mode` property will be used during the rendering. If its value is `mode="archive"` then only the archived ToDos will be shown. The `events` is an object which we will fill right away. That's the place where we place the DOM events mapping. The rest of the methods are responses of the user interaction and they are directly linked to the needed features. For example, `priorityUp` and `priorityDown` changes the ordering of the

ToDos. `archive` moves the item to the archive area. `changeStatus` simply marks the ToDo as done.

It's interesting what is happening inside the `initialize` method. Earlier we said that normally you will bind the changes in the model (the collection in our case) to the `render` method of the view. You may type `this.model.bind('change', this.render)`. But very soon you will notice that the `this` keyword, in the `render` method will not point to the view itself. That's because the scope is changed. As a workaround, we are creating a handler with an already defined scope. That's what Underscore's `bind` function is used for.

And here is the implementation of the `render` method.

```
// views/list.js
render: function() {
    var html = '<ul class="list">',
        self = this;
    this.model.each(function(todo, index) {
        if(self.mode === "archive" ? todo.get("archived") === true : todo.get("archived") ==
            var template = _.template($("#tpl-list-item").html());
            html += template({
                title: todo.get("title"),
                index: index,
                archiveLink: self.mode === "archive" ? "unarchive" : "archive",
                done: todo.get("done") ? "yes" : "no",
                doneChecked: todo.get("done")  ? 'checked=="checked"' : ""
            });
        }
    });
    html += '</ul>';
    this.$el.html(html);
    this.delegateEvents();
    return this;
}
```

We are looping through all the models in the collection and generating an HTML string, which is later inserted into the view's DOM element. There are few checks which distinguish the ToDos from archived to active. The task is marked as *done* with the help of a checkbox. So, to indicate this we need to pass a `checked=="checked"` attribute to that element. You may notice that we are using `this.delegateEvents()`. In our case this is necessary, because we are detaching and attaching the view from the DOM. Yes, we are not replacing the main element, but the events' handlers are removed. That's why we have to tell Backbone.js to attach them again. The template used in the code above is:

```
// index.html
```

```
<script type="text/template" id="tpl-list-item">
    <li class="cf done-<%= done %>" data-index="<%= index %>">
        <h2>
            <input type="checkbox" data-status <%= doneChecked %> />
            <a href="javascript:void(0);" data-up>&#8593;</a>
            <a href="javascript:void(0);" data-down>&#8595;</a>
            <%= title %>
        </h2>
        <div class="options">
            <a href="#edit/<%= index %>">edit</a>
            <a href="javascript:void(0);" data-archive><%= archiveLink %></a>
            <a href="#delete/<%= index %>">delete</a>
        </div>
    </li>
</script>
```

Notice that there is a CSS class defined called `done-yes`, which paints the ToDo with a green background. Besides that, there are a bunch of links which we will use to implement the needed functionality. They all have data attributes. The main node of the element, `li`, has `data-index`. The value of this attribute is showing the index of the task in the collection. Notice that the special expressions wrapped in `<%= ... %>` are sent to the `template` function. That's the data which is injected into the template.

It's time to add some events to the view.

```
// views/list.js
events: {
    'click a[data-up]': 'priorityUp',
    'click a[data-down]': 'priorityDown',
    'click a[data-archive]': 'archive',
    'click input[data-status]': 'changeStatus'
}
```

In Backbone.js the event's definition is a just a hash. You firstly type the name of the event and then a selector. The values of the properties are actually methods of the view.

```
// views/list.js
priorityUp: function(e) {
    var index = parseInt(e.target.parentNode.parentNode.getAttribute("data-index"));
    this.model.up(index);
},
priorityDown: function(e) {
    var index = parseInt(e.target.parentNode.parentNode.getAttribute("data-index"));
    this.model.down(index);
},
archive: function(e) {
    var index = parseInt(e.target.parentNode.parentNode.getAttribute("data-index"));
    this.model.archive(this.mode !== "archive", index);
},
changeStatus: function(e) {
    var index = parseInt(e.target.parentNode.parentNode.getAttribute("data-index"));
    this.model.changeStatus(e.target.checked, index);
}
```

Here we are using `e.target` coming in to the handler. It points to the DOM element which triggered the event. We are getting the index of the clicked ToDo and updating the model in the collection. With these four functions we finished our class and now the data is shown to the page.

As we mentioned above, we will use the same view for the `Archive` page.

```
list: function(archive) {
    var view = ViewsFactory.list();
    api
    .title(archive ? "Archive:" : "Your ToDos:")
    .changeContent(view.$el);
    view.setMode(archive ? "archive" : null).render();
},
archive: function() {
    this.list(true);
}
```

Above is the same route handler as before, but this time with `true` as a parameter.

Adding & Editing ToDos

Following the primer of the list view, we could create another one which shows a form for adding and editing tasks. Here is how this new class is created:

```
// App.js / views factory
form: function() {
    if(!this.formView) {
        this.formView = new api.views.form({
            model: api.todos
        }).on("saved", function() {
            api.router.navigate("", {trigger: true});
        })
    }
    return this.formView;
}
```

Pretty much the same. However, this time we need to do something once the form is submitted. And that's forward the user to the home page. As I said, every object which extends Backbone.js classes, is actually an event dispatcher. There are methods like `on` and `trigger` which you can use.

Before we continue with the view code, let's take a look at the HTML template:

```
<script type="text/template" id="tpl-form">
    <form>
        <textarea><%= title %></textarea>
        <button>save</button>
    </form>
</script>
```

We have a `textarea` and a `button`. The template expects a `title` parameter which should be an empty string, if we are adding a new task.

```
// views/form.js
app.views.form = Backbone.View.extend({
    index: false,
    events: {
        'click button': 'save'
    },
    initialize: function() {
        this.render();
    },
    render: function(index) {
        var template, html = $("#tpl-form").html();
        if(typeof index == 'undefined') {
            this.index = false;
            template = _.template(html, { title: ""});
        } else {
            this.index = parseInt(index);
            this.todoForEditing = this.model.at(this.index);
            template = _.template($("#tpl-form").html(), {
                title: this.todoForEditing.get("title")
            });
        }
        this.$el.html(template);
        this.$el.find("textarea").focus();
        this.delegateEvents();
        return this;
    },
    save: function(e) {
        e.preventDefault();
        var title = this.$el.find("textarea").val();
        if(title == "") {
            alert("Empty textarea!"); return;
        }
        if(this.index !== false) {
            this.todoForEditing.set("title", title);
        } else {
            this.model.add({ title: title });
        }
        this.trigger("saved");
    }
});
```

The view is just 40 lines of code, but it does its job well. There is only one event attached and this is the clicking of the save button. The render method acts differently depending of the passed `index` parameter. For example, if we are editing a ToDo, we pass the index and fetch the exact model. If not, then the form is empty and a new task will be created. There are several interesting points in the code above. First, in the rendering we used the `.focus()` method to bring the focus to the form once the view is rendered. Again the `delegateEvents` function should be called, because the form could be detached and attached again. The `save` method starts with `e.preventDefault()`. This removes the default behavior of the button, which in some cases may be submitting the form. And at the end, once everything is done we triggered the `saved` event notifying the outside world that the ToDo is saved into the collection.

There are two methods for the router which we have to fill in.

```
// App.js
newToDo: function() {
    var view = ViewsFactory.form();
    api.title("Create new ToDo:").changeContent(view.$el);
    view.render()
},
editToDo: function(index) {
    var view = ViewsFactory.form();
    api.title("Edit:").changeContent(view.$el);
    view.render(index);
}
```

The difference between them is that we pass in an index, if the `edit/:index` route is matched. And of course the title of the page is changed accordingly.

Deleting a Record From the Collection

For this feature, we don't need a view. The entire job can be done directly in the router's handler.

```
delteToDo: function(index) {
    api.todos.remove(api.todos.at(parseInt(index)));
    api.router.navigate("", {trigger: true});
}
```

We know the index of the ToDo which we want to delete. There is a `remove` method in the collection class which accepts a model object. At the end, just forward the user to the home page, which shows the updated list.

Conclusion

Backbone.js has everything you need for building a fully functional, single page application. We could even bind it to a REST back-end service and the framework will synchronize the data between your app and the database. The event driven approach encourages modular programming, along with a good architecture. I'm personally using Backbone.js for several projects and it works very well.

Krasimir Tsonev

Krasimir Tsonev is a coder with over ten years of experience in web development. With a strong focus on quality and usability, he is interested in delivering cutting edge applications. Currently, with the rise of the mobile development, Krasimir is enthusiastic to work on responsive applications targeted to various devices. Living and working in Bulgaria, he graduated at the Technical University of Varna with a bachelor and master degree in computer science. If you'd like to stay up to date on his activities, refer to his blog or follow him on Twitter.

KrasimirTsonev

Weekly email summary

View on Github

Translations

Envato Tuts+ tutorials are translated into other languages by our community members—you can be involved too!

Translate this post

Powered by native

25 Comments **Tuts+ Hub** 1 **Login**

Recommend 1 **Share** **Sort by Best**

Join the discussion…

Alejandro Exojo • 2 years ago

I don't want to sound rude, but if the backbone world needs something right now, is certainly not the n-th basic introduction to Backbone that uses a To-Do application as example. There are dozens of those. I've been searching extensively for good documentation for Backbone for the usecase of building something really big, and all I can find are lots of small examples repeating the same boilerplate. My conclusion is that, unless you are a seasoned Backbone developer, going with Backbone alone is suicidal if you want to build something large. Is great to provide some unopinionated base layer, but if you are a newbie, you'll need an experienced opinion, and you'll better check Marionette or LayoutManager at least.

Also, I think is a really bad approach to use "trigger: true" when calling navigate() on the router. And I'm not the only one.

22 • Reply • Share ›

MadMax11 Alejandro Exojo • 2 years ago

I agree. I understand to do lists are great for showing crud operations. Hower let's please get to the point where a basic skeleton app has users / authentication / routing / crud / remote data sync and possibly testing. This might sound a lot but they are the minimum requirements for almost all use cases and they are not trivial when business logic has gone to the front end.

1 • Reply • Share ›

Jeffrey Briceño Alejandro Exojo • 2 years ago

I'm agree.

The backbone oficial documentation has many large projects examples like Disqus, there is no doubt that Backbone is wonderful, but a TODO list for tutorial is kinda insipid.

Addy Osmani in his book Backbone fundamentals has the same project, but mush more powerful.

1 • Reply • Share ›

Wilberto Casillas Alejandro Exojo • 10 months ago

Perhaps you are right but in defense of the author, TutsPlus might have been lacking in a backbone tutorial. I also thought the link to the JS Design patterns was useful.

• Reply • Share ›

leoromanovsky Alejandro Exojo • 2 years ago

There is really not much else to Backbone. One of the appealing things about it is that it's fairly lightweight, compared to other frameworks like Angular.

• Reply • Share ›

johndurbinn Alejandro Exojo • 2 years ago

Or just use React like most sane people do these days and leave this insane shit behind

• Reply • Share ›

Rommel Castro • 2 years ago

javascript:void(0); ????? really

5 ^ | v • Reply • Share ›

Just_Some_Nobody ➦ Rommel Castro • a year ago

Why not?

^ | v • Reply • Share ›

John D • 2 years ago

it is very bad to do app)

5 ^ | v • Reply • Share ›

Филипп Риговaнов • 2 years ago

link to github repository must be fixed

2 ^ | v • Reply • Share ›

Gábor Soós • 2 years ago

why not camelcase namespaces too?

2 ^ | v • Reply • Share ›

Tony Brown • 2 years ago

Link to the repo if anyone needs it
https://github.com/tutsplus/si...

3 ^ | v • Reply • Share ›

Unhappy • 2 years ago

Pretty poor tutorial, not consistent in what code is suppose to be followed, copied or just shown as an example. Lots of coding errors and bad practices. I recommend using backbones website instead of this guide.

1 ^ | v • Reply • Share ›

Michał Zgliczyński • 2 years ago

ProTip: Instead of rewriting the swap in the down function, you could just reuse the up method. Moving down the index-th element is the same as moving up the (index+1)-th element. with that:

```
down: function(index) {
this.up(index + 1);
},
```

Mistakes:
1) // collections/ToDos.js line: 8
there should be a comma at the end:
model: app.models.ToDo,

1 ^ | v • Reply • Share ›

Maik Diepenbroek • 2 years ago

Sidenote: Typo: delteToDo => deleteToDo.
On topic: Good starting point for learning the basic-basics of backbone but i agree with suy21, i appreciate the effort you put into this article but a real life example would be far more useful.

I have not doubt that your knowledge of backbone is great and you could write a real life example as well. Would be greatly appreciated!

1 ^ | ∨ • Reply • Share ›

SPeed_FANat1c • 2 years ago

So backbone is still good? From one good programmer, who has experience with backbone and angularJS - he says learn angular and thats it, its way better according to him. I also have read that angular has two way binding, while backbone has one way binding , so probably that is big thing about angular and so backbone is one step back.
When you want to build something bigger than todo app like this. For small applications you can do fine with jquery alone. I started experimenting with angular examples. Not sure now should I focus on angular or get better at backbone? With backbone I have bigger knowledge than with angular.

1 ^ | ∨ • Reply • Share ›

Frederik Krautwald ➦ SPeed_FANat1c • 2 years ago

Actually, two-way binding causes major problems. Facebook knows this. So they created flux https://facebook.github.io/flu...

^ | ∨ • Reply • Share ›

SPeed_FANat1c ➦ Frederik Krautwald • 2 years ago

now the same guy tries out react.js, says angular is slow on phones :D so kind of you use so much time to learn new framwework, as we know angular is one of the harder frameworks, just to find out that it kind of sucks. Ok maybe it is ok, in some cases. He also claimed that even on desktop computers it lags when there is lot of data.

^ | ∨ • Reply • Share ›

Teguh Kusuma • 3 months ago

I am so confused with this tutorial.

^ | ∨ • Reply • Share ›

Disquser_Hard • a year ago

Thank you for such useful info!

Let me recommend the laconic and informative article for those who are interested in.

^ | ∨ • Reply • Share ›

Jake Wilson • a year ago

So.... I think you forgot to link or paste in the CSS.... kind of an important part.

^ | ∨ • Reply • Share ›

Jake Wilson • a year ago

The one major negative thing that stands out about this tutorial is that it doesn't give you any spots along the way to stop and view the current status of your page. It's hard to learn in an iterative fashion if you can't see what your code additions actually do. It's just this huge "write this 200 lines of code" and then see your result. Not a good way to learn.

^ | ∨ • Reply • Share ›

Adam Buczek • a year ago

Shameful edit.

Reply • Share ›

Sebastian Junior • 2 years ago

Well, I learned backbone last week so for me, it was a good tutorial. Tho I still haven't actually got it to work just yet. Working out the bugs!

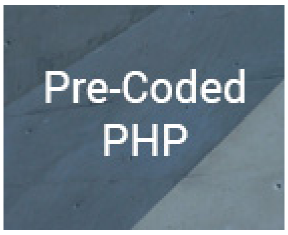
Reply • Share ›

zhrivodkin • 2 years ago

Bad tut! There are many mistakes (delEte in function name, commas in ToDos, not linked views in index.html, not optimal code, Backbone.history.start() missing).

Reply • Share ›

Check out Envato Studio's services



Browse PHP on CodeCanyon