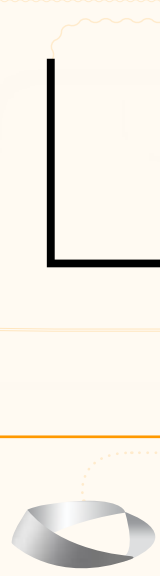
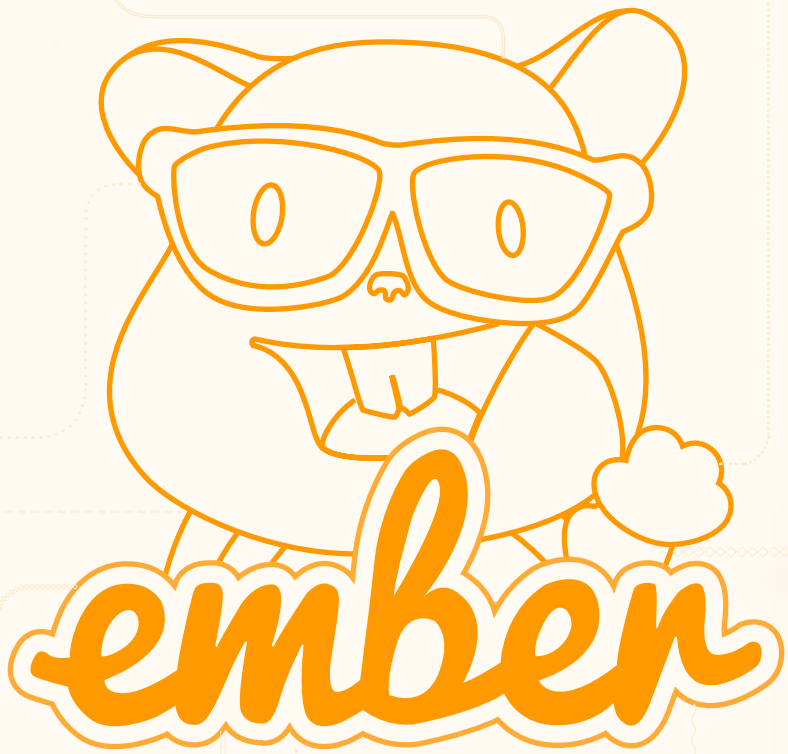
Ember Run Loop

by Jakub Niechciał



netguru



edition 1.0

Table of Contents

1 / Intro

2 / How do JavaScript frameworks actually work?

3 / Why does the Ember Run Loop exist? </>

4 / The Ember Run Loop from the inside

5 / Run, run loop, run - a few examples

- a Schedulers
- b Wrappers
- c Examples based on Ember 1.7 - for more complex code tricks </>
  - i Nested run loops </>
  - ii Scheduling algorithm </>
- d Ember.run.sync() </>
- e Observers vs. computed properties </>

6 / Handling external events with the Ember Run Loop </>

7 / Autoruns and their implications </>

8 / Summary

9 / About us

Thanks for downloading our guide! We'd love to stay in touch - you can find us here:

1 / Intro

The Ember run loop is one of the most interesting mechanisms in the Ember framework. Unfortunately, it is not well documented in official guides. The run loop might seem like a well hidden abstraction, but, when working on complex Ember projects, you'll notice it when you:

- bump into strange cases where data is synchronized in a weird order,
- find actions executing in unexpected ways and, most importantly,
- notice that your vanilla JS or jQuery plugins are not working

While working with Ember, I found myself frequently searching the web for the Ember run loop content. You can surely find some interesting posts out there, but here I'd like to share a summary of an Ember Run Loop webinar that I held for the Netguru team.

In this guide, I will cover why the run loop exists and how it actually works. Then, I will share a couple of interesting examples, all of which are available in JSBin so you can work with them yourself. We will also see how to use the run loop while wrapping external JS libraries and explore the autorun mechanism.
At the end, I've also included some other interesting resources that helped me understand the run loop.

2 / How do JavaScript frameworks actually work?

To answer this question you must understand more deeply how every JS framework works at the low level, without all the abstractions. First of all, when you load the website with its JS code, the whole code, every line, is executed only once - right after the download finishes. For this reason, basically all the code that works in our browsers is event-driven - it works by responding to some events that happen in the environment. These events are fired by the browser and handled by handlers provided by your scripts. Crucially, browsers execute at most one event per one millisecond. This limitation may be considered as both an advantage and a disadvantage. We'll talk about this again later.

The Ember framework consists of a very short setup phase in which it registers handlers for multiple events, including ***keyUp***, ***keyDown***, ***mouseMove***, and others. Feel free to check out the [full list of handlers], too.

This setup phase is executed as a handler for the `DOMContentLoaded` event to make sure that the full content of a page is ready. After this setup phase, everything that happens in our single page application is a response to user behaviour

3 / Why does the Ember Run Loop exist?

Before you find out how the run loop works, let's stop for a while and talk about why it's necessary. If you understand why it exists, the underlying mechanism should become much clearer.

The main reason for creating the run loop is to improve the performance of our applications. The run loop reduces the number of expensive actions, such as rendering, by batching them in queues. Moreover, it organizes the execution of our code in logical blocks, so it's easier to maintain and we can have more control over the order of execution.

However, to achieve this you need to know how the run loop works and how to use it properly. And this is the hard part.

Lets look at the following example from the Ember Guides.

Example #1

```
<div id="foo"></div>
<div id="bar"></div>
<div id="baz"></div>
```

```
foo.style.height = "500px" // write
foo.offsetHeight // read (recalculate style, layout, expensive!)

bar.style.height = "400px" // write
bar.offsetHeight // read (recalculate style, layout, expensive!)

baz.style.height = "200px" // write
baz.offsetHeight // read (recalculate style, layout, expensive!)
```

In this very academic example, you set the height of the DIV elements and calculate their offset one after another.
It demands three very quick operations of setting, and three more expensive operations of calculating layouts and offsets.
If you could batch them by similarity, you could get a huge performance gain, due to caching the values of offsets and only having one layout recalculation.

Example #2

```
foo.style.height = “500px” // write
bar.style.height = “400px” // write
baz.style.height = “200px” // write

foo.offsetHeight // read (recalculate style, layout, expensive!)
bar.offsetHeight // read (fast since style and layout is already known)
baz.offsetHeight // read (fast since style and layout is already known)
```

This example is pretty rare in day-to-day work. Let’s take a look at a more relevant Ember-style example that consists of one computed property based on two attributes - `firstName` and `lastName` . Somewhere in your code, potentially in some action, you set these two one after another.

Example #3

```
 {{user.firstName}}
 {{user.fullName}}
```

```
user: Ember.Object.create({firstName:’Tom’, lastName:’Huda’});

fullName: Ember.computed ‘user.firstName’, ‘user.lastName’, function() {
    this.get(‘user.firstName’) + ‘ ‘ + this.get(‘user.lastName’);
});

user.set(‘firstName’, ‘Yehuda’);
// {{firstName}} and {{fullName}} are updated

user.set(‘lastName’, ‘Katz’);
// {{lastName}} and {{fullName}} are updated
```

Without any batching mechanism you would end up recalculating a computed property twice during one action. This is obvious waste of time. Indeed, if you had a scenario where the rendering mechanism had a higher priority than computed properties, but lower than setting attributes, you would end up re-rendering the layout four times!
The run loop lets you avoid such horrible messes.

I'll now answer the most important question - **what exactly is Ember run loop?** Well, it's not a loop in the sense of the common for-loop or while-loop. Rather, **it's a mechanism that batches assorted actions** (like setting, actions, or transitions) and then decides to execute them in some planned order.

So, when does the run loop start batching? As I said earlier, **Ember is a fully event-driven framework**. Everything that happens is a reaction to user behaviour. The user has clicked something, has moved the mouse over the page or pressed some key. Each time the lowest level Ember handler starts handling the event, a run loop is created and starts accepting jobs. From the low-level handler, **the stack of execution runs up through all the abstraction layers of Ember**, eventually reaching the code we ourselves have written.

You can, for example, set some attributes and perform a transition using the `set` and `transitionToRoute` methods.
The implementation of these methods uses the run loop by scheduling synchronization of bindings used by these attributes and the transition. Basically, nearly everything you do using the Ember API is scheduled into the run loop and left to execute in the future. After the stack of execution gets back to the low level handler, the run loop is closed.

Ok, so **when is the run loop executed?** Right after it is closed, in the same event handler. If you remember what I said earlier about the event environment in browsers, you'll recall that the run loop can execute at most every one millisecond.
This is an important fact that I will return to later.
It also means that Ember's reaction to user behaviour is fully enclosed in the run loop and is executed before the next event arises.

4 / The Ember Run Loop from the inside

Next question - how are these jobs executed? The internals of the Ember run loop consist of six different queues that are ready to accept jobs. Each is responsible for a different kind of job:

- `sync` for synchronization of bindings (e.g. between controller and components in the template)
- `action` for handling actions and promises
- `routerTransitions` for transitions
- `render` for rendering templates
- `afterRender` for any job that must be performed after rendering
- `destroy` for handling garbage

I've listed them in order of execution. However, this order is not so straightforward. After you sync the bindings in the first queue, it is very likely that our action code will generate new bindings that will be batched to the run loop sync queue and thus skip execution. If the algorithm simply executes the queues in the order they are listed, until all of them are empty, we won't get the performance improvement that we want - rendering will still happen multiple times.

Therefore, the algorithm for execution returns to the first queue after flushing each of them and checks for any new jobs. Of course, there still might be some jobs in the `afterRender` or `destroy` queues that will break the concept and add some bindings, leading to re-rendering the DOM again in the same run loop. But this is very uncommon and is mostly the result of bad code or intended behaviour - we will return to this later.

I hope that the foundations of the Ember run loop mechanism are now much clearer. In the next section I will cover some examples that will clarify the concepts I described.

5 / Run, run loop, run - a few examples

The Ember run loop provides a very interesting API which can give you full control over your code execution.
Firstly, I will cover some of the most common methods used from the Ember Run namespace. I've intentionally divided them into two blocks - **schedulers and wrappers**.

The schedulers provide the ability to schedule passed functions in the existing run loop, while the wrappers wrap the passed function in a completely new instance of Ember run loop. Yup, that's not a typo - **run loop is not a singleton** and Ember can have multiple run loop instances at once. However, each opened instance blocks the execution of its parent until all of its queues are flushed (examples coming up soon!).

A / Schedulers

- `Ember.run.schedule` allows you to schedule a passed function to the exact queue pointed out in the first argument.

- `Ember.run.once` allows you to schedule a passed function (cannot be anonymous!) by default to the `actions` queue and makes sure that this function won't be executed more than once in the current run loop instance.

- `Ember.run.debounce` works the same as jQuery debounce but is run loop compliant. Executes a passed function after a specified time and resets the timer every time it's called again. This means that the passed function won't be executed in the current run loop - the minimum time that can pass is one millisecond. However, it will be executed in future and all of its Ember API methods will be properly scheduled. Please note that the passed function cannot be anonymous!

- `Ember.run.throttle` allows you to execute a passed function immediately and ensures that during a defined period of time it won't be executed again in any existing run loops. Again, the passed function cannot be anonymous!

B / Wrappers

- Ember.run just wraps the passed function in a new instance of Ember run loop. It freezes execution of the current run loop until all the jobs queued during execution of the passed function are flushed.

- Ember.run.next wraps the passed function in a new instance of Ember run loop that will be executed after one millisecond (again, the smallest period of time between handling events), i.e., the next possible run loop.

- Ember.run.bind is a very powerful method that is used to embrace external JS libraries. It takes the passed function and context and returns a function that, when executed, will execute the passed function with proper context and wrapped by a new instance of Ember run loop.

Last, but not least, I would like to share with you one private Ember API method that might be useful. `Ember.run.sync` is a method that explicitly makes the current Ember run loop completely flush the `sync` queue. This is synchronous execution and in the next line of code, the `sync` queue will be empty, so you can be sure that all the bindings are in the proper places.
This method can be useful, but remember that private is private and it could easily be changed in the future.

C / Examples based on Ember 1.7 - for more advanced code tricks

The next few examples will be based on Ember 1.7 - quite an old version. However, this Ember distribution is not a regular one, but modified to log every interesting thing about the run loop - when it starts, what it does and when it ends. This implementation is prepared by @eoinkelly - check out his full tutorial about the run loop. I prepared a JSBin with this working noisy run loop, with mouse move events removed from run loop triggers. Take a look if you want to try out some more complex ideas.

C / i / Nested run loops

To help you better understand how the run loop is just a regular object, instantiated on demand, imagine how nesting run loops might look like in the following situation:

Example #4

```
this.$().click(function() {
  Ember.run(function () {
    Ember.debug('In my own runloop');
    $('body').css('background-color', 'pink');
    Ember.run(function () {
      Ember.debug('In a nested runloop');
      $('body').css('background-color', 'red');
    });
  });

  Ember.run(function () {
    Ember.debug('In another of my own runloops');
    $('body').css('background-color', 'yellow');
  });
});
```

When the user clicks anywhere on the controller template, we open a new run loop instance and pass an anonymous function which logs that it started executing. Then we open the next run loop and do the same.

To understand how this works, it’s helpful to use a noisy run loop and let the run loop tell us what it does.

Fin out how it works Open in JSbin 

It turns out that opening a new run loop inside the other one freezes the parent’s execution. As soon as all the queues in the nested run loop are flushed, execution returns to the upper run loop. Notice that **when the stack enters the nested run loop, almost nothing from its parent was executed** - all the code was just queued, waiting to be executed. Don’t forget about this - it can cause headaches!

C / ii / Scheduling algorithm

I have analyzed the algorithm that underlies the Ember run loop. This algorithm, though simple, is an extremely important part of the mechanism. It tries to make sure that rendering (the most expensive action) is executed only once. However, there might be situations where you need to make sure that something happens after rendering.

Your actions after rendering may introduce new bindings which should be synced. Let’s look at how this works out in a simple example:

Example #5

```
actions: {
  scheduleTasks: function() {
    Ember.run.schedule(‘afterRender’, this, function()
      console.log(“CUSTOM: I’m in afterRender”);
      Ember.run.schedule(‘sync’, this, function() {
        console.log(“CUSTOM: I’m back in sync”);
      });
    });
  }
}
```

As you can see, in the action handler you schedule an anonymous function into the afterRender queue, which, in turn, schedules back to the sync queue. If you go to a live demo below, you can see in the console what actually happens. The run loop, after initially flushing all the queues from the sync to afterRender, goes back to the sync queue. Don’t forget - the Ember run loop is safe to schedule in any queue. You can be sure your job will be done.

Scheduling example Open in JSbin 

D / Example of using Ember.run.sync()

The following example is tricky. It does not adhere to the *data-down-action-up* convention and it's not good practice for your project. But it's a great illustration of how sync works.

Let's imagine you have a component with an input field. You use jQuery to handle a change event on that input. In that handler, properly wrapped in a run loop, you set an internal component value and send the action up to the controller that it's inside.

Example #6

```
App.SomeValueComponent = Ember.Component.extend({
  valueChanged: function(val) {
    this.set('value', val);
    this.sendAction('valueChange');
  },

  didInsertElement: function() {
    this.$("input").on("keyup", Ember.run.bind(this, function() {
      this.valueChanged(this.$("input").val());
    }));
  }
});
```

The controller handles this action and logs the mentioned component value through a binding called `value`.

Example #7

```
App.IndexController = Ember.Controller.extend({
  actions: {
    handleChange: function() {
      console.log(this.get("value"));
    }
  },
});
```

What happens after each change and what is the log output from the controller? If you take a at an excample below, you'll see that after each key click, the log output lags by one character.

Ember.run.sync() example Open in JSbin 

Why? To understand what's going on you have to look in the Ember codebase, in particular - its implementation of `sendAction`. It turns out that `sendAction` is executed synchronously, right after it shows up in the code.

This is a little bit confusing. Even though the code which it is inside is wrapped in Ember queues, **the console log is immediate and is always late one run loop iteration** because of the still-unsynced binding between component and controller.

How can you bypass this problem? Well, you could send that value as an argument of that action and take it from the args rather than from the bindings. But I promised to show you the sync method. Let’s add sync execution in controller and see what happens.

Ember.run.sync() example Open in JSbin 

As you can see, after syncing the bindings there is no delay between the log and the actual value. Again, please note that this design is not a good solution for your projects.
Use sync only when the design itself cannot be changed.

E / Example of using Ember.run.debounce

Sometimes you would like to execute some actions after a repeated event stops firing - like scrolling or typing input. The Ember run loop provides us with a very simple and convenient method called `debounce` that works the same as jQuery debounce, but is compliant with the run loop. It fires the passed function after the passed time period. If the debounce is called before the time passes, the timer is reset. Check fully working example below.

Ember.run.debounce example Open in JSbin 

F / Observers vs. computed properties

I have a tricky question for you: what is executed first in the Ember run loop - observers or computed properties? **Actually, observers are synchronous and are not queued into the Ember run loop.**

They are executed right after the variable change and will always be executed earlier than any computed property.
Let's take a look at a simple example with an almost identical observer and computed property:

Example #8

```
partOfNameChanged: Ember.observer("firstName", "lastName", function() {
    console.log("[Observer]: Executing...");
})

fullName: Ember.computed("firstName", "lastName", function() {
    console.log("[Computed property]: Executing...");
    return (this.get("firstName") + " " + this.get("lastName"));
})

toggleName: function() {
    this.set("firstName", "Foo");
    this.set("lastName", "Bar");
}
```

How many times will you see the observer and computed property logged? Check your answer! What's happening is that the observer is fired twice, once when `firstName` is changed and once when `lastName` is changed. However, the computed property will be fired only once as both of these changes happen in one instance of run loop and will be queued and evaluated before calculating the computed property.

Let’s make use of what you’ve learned so far. Suppose we need the observer to execute some action on change, but only once in each run loop. We could do it in a computed property, but that would be inconsistent and against convention. Computed properties are for computing, while observers are for reacting to changes.

Let’s use the `Ember.run.once` method:

Example #9

```
partOfNameChanged: Ember.observer(“firstName”, “lastName”, function() {
    Ember.run.once(this, “doSomeProcessing”);
})

doSomeProcessing: function() {
    console.log(“[Observer]: Executing...”);
}

fullName: Ember.computed(“firstName”, “lastName”, function() {
    console.log(“[Computed property]: Executing...”);
    return (this.get(“firstName”) + “ “ + this.get(“lastName”));
})

toggleName: function() {
    this.set(“firstName”, “Foo”);
    this.set(“lastName”, “Bar”);
}
```

Check out live demo below to see the results. It’s a perfect solution for using observers in a way compliant with the run loop. Their synchronous nature is both a blessing and a curse. However, with our toolset of run loop methods we can use this to our advantage.

Observers example Open in JSbin 

6 / Event handling outside the Ember Run Loop

Until now, it seems that you don't need to worry too much about the run loop. It's always opened on handling a user event and all our actions are executed within it.
However, **there are situations when Ember is not handling events, but you.** These can be custom event handling (like in the former example with sync) or registration of callbacks for AJAX calls or any other asynchronous callback.

In such cases, you simply wrap all the code to be executed in an Ember.run instance. Which isn't very difficult. On the other hand, there may be situations where you want to pass your controller or component methods to be passed as callbacks to some external jQuery plugins.

If you pass them as they are, first of all they won't be executed in a run loop and second of all, they won't have a proper `this` scope (and you want your controller or component to be `this` scope, of course).
To do so, we can pass a binded method using `Ember.run.bind` - it both executes the method inside a run loop and binds `this` scope to the controller. Take a look at this snippet from @iStefo - select2 ember wrapper:

Example #10

```
this._select = this.$().select2(options);

// run ember bindings on after select2 `change` event
this._select.on("change", run.bind(this, function() {
  var data = this._select.select2("data");
  this.selectionChanged(data);
}));
```

Ok, but what happens if you don't wrap your code in a run loop?

7 / Autoruns and their implications

To rephrase the question - can we schedule jobs to the run loop while it is not running? Take a look at following snippet:

Example #11

```
$("a").click(function() {
  console.log("Doing things...");

  Ember.run.schedule("actions", this, function() {
    // Do more things
  });
});
```

The answer is yes, you can. **Ember has a mechanism called autorun that initiates the run loop if you try to schedule anything while it's not running.** This simple, but powerful mechanism has its disadvantages too. If you totally skip explicitly starting run loops, all of them will be opened by autorun.

However, in a testing environment, all asynchronous helpers (like click, fillIn, visit, etc.) will wait for all run loops to flush before going further (e.g. to asserts). If autoruns, in some edge cases, don't cover all the code being executed, you end up with hard-to-debug-problems that only appear in a testing environment.
To prevent this, run loops are switched off in testing mode, which forces you to explicitly start run loops everywhere you have tests written (hopefully throughout the entire app!). This improves your experience while working in the development mode.

Take a look at a live demo below which shows what happens in testing mode while handling custom events. Remove the first line `Ember.testing = true;` and you will return to the development mode, where everything works fine.

Autorun example Open in JSbin 

The next example is the same, but with the handler wrapped in a run loop. It works in all situations, regardless of what mode is active.

Handler wrapped in a run loop example Open in JSbin 

How is autorun activated? Nearly every Ember API method is compliant with the run loop and internally schedules its respective job `set` schedules setting and bindings, `transitionTo` schedules a transition, etc. The scheduling methods that are used in the private API are the same as the public `Ember.run.schedule` method. Reading the implementation you see that schedule checks if any autorun is running by making use of the internal run loop references counter.
If not, it starts a new one. Fairly easy. This example and another one will show you the topic more thoroughly.

8 / Summary

It was quite a long journey through the Ember run loop. The main points that I hope everyone takes away from this are:

- The Ember run loop is a mechanism to batch and defer actions, then execute them optimally
- The Ember run loop is a regular object, not a singleton and we can have multiple run loops simultaneously
- Be aware of asynchronous executions of Ember API methods
- Everything you do in custom handlers should be wrapped in a run loop.

Of course, there is a lot more to know about the run loop. Most importantly, I did not mention the implementation details.
If you would like to know more about the Ember Run Loop, you can check out the following links:

- Ember Run Loop in the Ember Guides
- Ember Run Loop API Reference

- Ember Run Loop Handbook
- Backburner.js and the Ember Run Loop
- What is the Ember Run Loop and how does it work?

Finally, you can check out the presentation that I prepared for the internal webinar at netguru, on which this guide was based. It took me a lot of time to understand the run loop and I want to share this knowledge.

9 / About

Jakub Niechciał

Jakub has obtained a Master's degree at Poznań University of Technology in Control Engineering and Robotics. During the studies, he dealt with computer vision and machine learning. He also spent almost two years working as a front-end developer using CSS and jQuery, but eventually skipped to Ruby on Rails and stayed in back-end for real. In 2014, while working at Netguru, he discovered Ember.js and React.js and has found a lot of fun diving deep into these two frameworks. He loves biking, watching HBO's TV-series and drum'n'bass music.

netguru

We are an Agile web and mobile development team. Netguru web developers build lean and beautiful websites and applications for early-stage startups to major corporations around the world. For any ordered project, we also provide design services with the best UX practices in mind. Our goal is to create experience that delivers outstanding results on each medium.
As an employer of more than 130 awesome folks, we believe in transparency and flexibility. Our teams work remotely from more than 10 international locations. We are in the constant search for the best specialists to join us in developing new exciting projects.
Hope you enjoyed your read! Now join us:

Sign up for our monthly Newsletter

Once a month we send out a free newsletter with a roundup of startup, design and web dev tips, tricks and resources curated from across the web.

Join more than 5000 subscribers!

Subscribe Me

netguru