Stimulus.js / Ruby on Rails

PAINLESS JavaScript with RAILS

HOW TO BUILD INTERACTIVE APPS WITH RAILS AND JAVASCRIPT EFFECTIVELY

BY PAWEŁ DĄBROWSKI

Introduction

Even if you don't like JavaScript, you can't deny that this technology is present everywhere in modern web applications. It also successfully influences the way we are building the applications, including the Ruby on Rails framework.

This ebook is a quick and painless introduction to the Stimulus.js framework that will help you add JavaScript to your Rails application and keep creating modern web applications without losing the joy of writing beautiful and straightforward code.

What you will learn from this ebook

After reading this ebook, you will be able to:

- Tell what Stimulus.js is, how it compares to other JavaScript frameworks, and why it's beneficial to use it in your Rails applications
- Add Stimulus.js framework to the new Rails application as well to the existing legacy applications
- Write applications with Stimulus.js and Ruby on Rails framework
- Search for helpful information about the framework itself and the community around it

Is this ebook up to date?

Code presented in this ebook comes from the latest stable versions of the Ruby on Rails framework and Stimulus.js framework. The contents are also updated frequently when a new stable version of frameworks is released, so you don't have to worry about that.

Right now, the contents were created with the usage of the following versions:

- Ruby: 3.0.1
- Ruby on Rails: 6.1.3.1
- Stimulus.js: 2.0.0

Background information about Stimulus

This chapter is a quick introduction to the idea behind the Stimulus.js framework and its architecture. Reading it before diving into the code will allow you to better understand this library's general concepts and approach. You would also know if implementing Stimulus in your application will be beneficial.

What is Stimulus.js

Unlike other more extensive frontend frameworks that you may know, Stimulus was not designed to take complete control over your frontend part of the application. It exists to help you extend your Rails application with the JavaScript code to make it more interactive, so you don't have to worry about rewriting your views or creating API endpoints to deliver the data to the front.

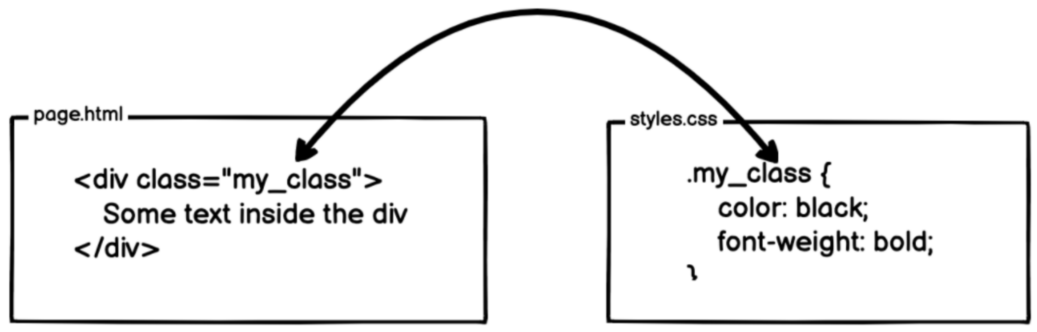
How Stimulus works

Stimulus's architecture is quite simple as there are no magical and advanced routers that select when to trigger a given code. If you are working with Rails, then you are already familiarized with **controllers**. In Ruby on Rails, a single controller is a class that accepts the request and decides what to do next; render the view, redirect or perform any other action.

In Stimulus is the same, the **controller** is responsible for the decisions, and it's just a JavaScript object that contains some logic for managing the HTML code of your code.

How stimulus connects the JavaScript controller to the view

As you may know, we can style the given HTML element by using a particular class or id attribute to which we refer in the CSS styles:

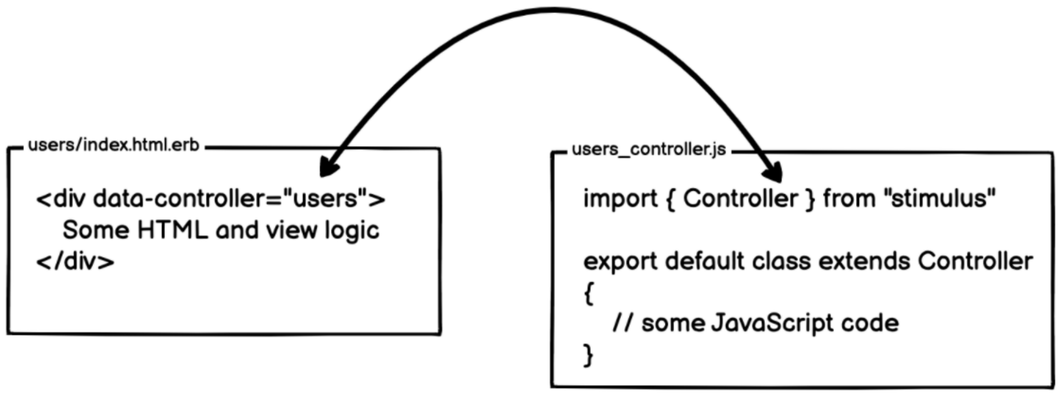


```
<div class="my_class">
  Some text inside the div
</div>
```

```
.my_class {
  color: black;
  font-weight: bold;
}
```

In the above example, the text inside the div will be in black and will be bold. The class attribute is the bridge between the HTML and CSS code in your website.

In Stimulus, the data-controller attribute is the bridge between the HTML and JavaScript code in your application:



```
<div data-controller="users">
  Some HTML and view logic
</div>
```

```
import { Controller } from "stimulus"

export default class extends Controller
{
  // some JavaScript code
}
```

Thanks to the data-controller attribute, Stimulus knows which controller to load when the given view is rendered. I will cover the topic of autoloading controllers a little bit later, so you don't have to worry about that right now.

The main concepts of the architecture

The controller is the entry point in the Stimulus' architecture. A single controller is a JavaScript object, and it contains other elements to manage the HTML code and events:

Actions

A single controller can define multiple actions - just like in Rails. Each action is a set of instructions that should be executed when a given event is triggered.

In Rails, we would say that the event is the GET or POST request, while in Stimulus, we can say that clicking on the button or submitting the form is an action.

Targets

You can treat the target as a single element in the HTML code, for example, div, input, or button. Each target has a name to which you can refer in your JavaScript controller to access and manipulate elements in the view easily.

Values

Values allow you to read attributes' values, write the attributes' values and observe changes in those values. For example, you might be interested in the input's value and perform some action when its value is changed.

Installation process

The installation process is fast and easy. Stimulus is available as an npm package, but it’s also possible to load it using the script tag.

Installation with Rails 4 or 5

Installation with Sprockets

Download the latest version of the library and put it inside the the vendor/assets/javascripts directory, which sprockets will autoload:

```
wget -O vendor/assets/javascripts/stimulus.umd.js
https://unpkg.com/stimulus/dist/stimulus.umd.js
```

Now to load the file, open the app/assets/javascripts/application.js file and import the file with the Stimulus library:

```
//= require stimulus.umd
```

You can now test if your installation was successful. Open again the application.js file where you required the library and put the following code:

```
(() => {
 const application = Stimulus.Application.start()

 application.register("hello", class extends Stimulus.Controller {
   connect() {
     console.log("Hello, Stimulus!", this.element)
   }
 })
})()
```

Open one of your views and put the div with the data-controller attribute with hello as value:

```
<div data-controller="hello">
 Hello!
</div>
```

Open the page, and you should see the debug output in the browser console.

Installation without Sprockets

If you would like to install Stimulus without Sprockets, you can include the source file directly in your layout:

```
<%= javascript_include_tag
"https://unpkg.com/stimulus/dist/stimulus.umd.js" %>
```

The next step is to initialize the Stimulus controller in the page source:

```
<script>
 (() => {
   const application = Stimulus.Application.start()
    application.register("hello", class extends Stimulus.Controller
{
     connect() {
       console.log("Hello, Stimulus!", this.element)
     }
   })
 })()
</script>
```

You can now update the view code to trigger the controller:

```
<div data-controller="hello">
 Hello!
```

```
</div>
```

Installation with Rails 6

If you would like to load Stimulus without Webpack, please take a step back to the previous paragraph where I demonstrated how you can load the library directly into your website using the script tag.

Webpacker library is a bridge between our Rails application and Webpack, and it comes with a handy command for installing Stimulus:

```
bundle exec rails webpacker:install:stimulus
```

If you are building a brand new application, make sure you installed webpacker before running the above command:

```
rails webpacker:install
```

That's it; you can now use Stimulus with your application. The install command created the app/javascript/controllers directory with example controller. You can now add your controllers there, and they will be autoloaded into the app.

It's time to build a real application with Stimulus!

Building with Stimulus.js and Rails

This tutorial will show you how you can quickly build an interactive application with real-time data processing. We will create a simple application where users can add a task to the to-do list and see tasks added by other users in real-time.

I will use Rails for the backend, Stimulus.js framework for the frontend, and ActionCable for web sockets. Sometimes creating and maintaining apps with a more complex frontend can become a nightmare for Rails developers; not this time!

Generating new Rails application

Make sure you have the latest version of Ruby and Rails installed in your system. I'm going to use Ruby 3.0.1 and Rails 6.1.3.1. You will also need Node; I used the latest recommended for regular users version, which is 14.16.0, when writing that book.

If you have all the required elements installed in your system, we can generate a brand new Rails application:

```
rails new todo -d=mysql
cd todo/
```

I used MySQL for the database but feel free to use any database you want. I won't use any database-specific code.

Stimulus installation

When the installation is done, the next step of the configuration is to install stimulus:

```
bundle exec rails webpacker:install:stimulus
```

Database creation

The last step of the configuration process is to create the database. Update the config/database.yml file with your database credentials and create the databases:

```
rake db:create
```

Generating the model

Our application will be a simple but interactive, to-do list application, so for demonstration purposes, we would need only one model: Task. One task can have a title and completion status. Let's generate the model and migrate the data:

```
rails g model Task title:string completed:boolean
```

Before we migrate the data, we have to make one change to the migration file. We have to set the completed field to be false by default:

```
class CreateTasks < ActiveRecord::Migration[6.1]
  def change
    create_table :tasks do |t|
      t.string :title, null: false
      t.boolean :completed, default: false

      t.timestamps
    end
  end
end
```

I also added null: false option to the title column to ensure that we won't save the task without the title. We will make the validation on the interface level as well, but it's always good to make the basic validation on the database level also.

Now we can migrate the database:

```
rake db:migrate
```

Adding test data

When building any application, it's always good to have some test data that we can use during the development process. Let's create then some seeds. Open db/seeds.rb file and add the following code:

```
Task.create(title: 'Clean the house', completed: false)
Task.create(title: 'Walk out the dog', completed: true)
Task.create(title: 'Wash the dishes', completed: false)
```

Now you can insert the data into the database:

```
rake db:seed
```

Creating tasks listing

The plan for the tasks listing is the following: we will create a controller for tasks that will serve the index action where all tasks will be listed.

We will also create a JavaScript controller with Stimulus that will load that list when visiting the root page of the application. We won't be using any fancy DOM updates, simple HTML injections. That's the reason why creating interactive applications with Stimulus and Rails is painless. Let's start!

The tasks controller

The first step is to create a controller:

```
touch app/controllers/tasks_controller.rb
```

What's essential, we don't want to render the layout when pulling the data from the tasks controller. Since we will inject the whole HTML parts, we would duplicate the code in the main layout.

For now, we need only the index action with basic tasks assignment so we can access them later in the view:

```ruby
class TasksController < ApplicationController
  layout false

  def index
    @tasks = Task.order('created_at DESC')
  end
end
```

The next step is to update the routes configuration in config/routes.rb file:

```ruby
Rails.application.routes.draw do
  resources :tasks, only: %i[index]
end
```

Our controller is now accessible, but the application complains about the missing views. Let's start with the index view:

```
mkdir app/views/tasks
touch app/views/tasks/index.html.erb
```

We will simply render the partial for each task record there:

```erb
<h1>Tasks:</h1>

<ul>
 <% @tasks.each do |task| %>
   <%= render 'task', task: task %>
 <% end %>
```

```
</ul>
```

and the task partial:

```
touch app/views/tasks/_task.html.erb
```

its source will be also very simple. We have to remember to strike the task title when it's completed:

```
<li>
 <% if task.completed %>
   <s><%= task.title %></s>
 <% else %>
   <%= task.title %>
 <% end %>
</li>
```

You can now start the server and visit http://localhost:3000/tasks address to see that our listing is rendered correctly.

The home controller

Our application does not yet have a controller that is triggered when the root URL is visited. I would call that controller a home controller with an index action:

```
touch app/controllers/home_controller.rb
mkdir app/views/home
touch app/views/home/index.html.erb
```

The source of HomeController is just a controller definition and the index action with the blank body:

```
class HomeController < ApplicationController
```

```
  def index

  end
end
```

However, the index view for the home controller will have some interesting HTML code:

```
<div data-controller="tasks" data-tasks-url-value="<%= tasks_path
%>" data-tasks-target='listing'>
 </div>
```

Let's break down the view structure and the data-* attributes meaning:

- data-controller - it informs the Stimulus framework what JavaScript controller should be automatically triggered when the view is rendered. In our case, it's tasks_controller.js that we will create in a while
- data-tasks-url-value - this declaration is a shortcut for the tasks controller to define a value variable. Such a variable is defined in the controller and has a default type added. We will use that variable to pull the list of tasks
- data-tasks-target - a target is an HTML element with which the Stimulus can interact. It could be a div, input, or anything else. You can later manipulate such element from the JavaScript controller level

Since you can operate on multiple JavaScript controllers in one view, the data-tasks-* pattern is used to separate attributes for each controller.

The last step for this part is to update config/routes.rb file:

```
Rails.application.routes.draw do
  root to: 'home#index'
  resources :tasks, only: %i[index]
end
```

The JavaScript controller

Finally, we can create the controller I talked about so much before:

```
touch app/javascript/controllers/tasks_controller.js
```

Put there the following definition:

```
import { Controller } from "stimulus"

export default class extends Controller {
  static targets = ['listing'];
  static values = { url: String }

  initialize() {
    this.load();
  }

  load() {
    fetch(this.urlValue)
      .then(response => response.text())
      .then(html => this.listingTarget.innerHTML = html)
  }
}
```

Because of the data-controller attribute, Stimulus knows that the tasks controller should be automatically loaded. When the controller is loaded, the initialize method is executed. In our case, we load the list of tasks from the URL defined in the data-tasks-url-value attribute and then apply it to the body of the listing target specified in the data-tasks-target attribute.

You can now visit the main URL http://localhost:3000/, and the list of tasks will be automatically loaded. We can now focus on adding the code for adding new tasks.

Adding new task

What might be surprising for you is that we don't have to update the JavaScript controller to implement a new task. Let's add the validation for the task's title presence first. Open app/models/task.rb file and put there the following line:

```
class Task < ApplicationRecord
  validates :title, presence: true
end
```

The next step is to update the TasksController and add create action that will handle the task creation request. It will simply save the record:

```
class TasksController < ApplicationController
  layout false

  def index
    @tasks = Task.order('created_at DESC')
    @task = Task.new
  end

  def create
    @task = Task.new(task_params)
    @task.save
  end

  private

  def task_params
    params.require(:task).permit(:title)
  end
end
```

I also made one addition to the index action. I saved a new Task object in the @task variable, so it's accessible from the index template to render the form partial.

Before we will create the form, let's update the routes to enable the create action:

```
Rails.application.routes.draw do
  root to: 'home#index'
  resources :tasks, only: %i[index create]
end
```

Now, it's time to create the form partial:

```
touch app/views/tasks/_form.html.erb
```

Put there the following content:

```
<%= form_for(task, remote: true, data: { action: 'ajax:success-
>tasks#load'}) do |f| %>
  <%= f.text_field :title %>
  <%= f.submit "Add" %>
<% end %>
```

There are two interesting elements in the form definition:

- remote: true option - thanks to this option, the application won't submit the form in a standard way. The ajax call will be performed
- data-action attribute - when the form submission will be successful, the load method from the tasks controller will be executed

Thanks to the above configuration, we will automatically pull an updated version of the tasks list if the new task is created successfully. Simple as that. You can now test the form, and you will see that when you add the new tasks, the list is updated immediately.

Marking a task as completed

This would be even easier to achieve than adding a new task. Again, we don't have to update a single line of JavaScript code to automatically mark the task as completed and automatically update the list.

Start with updating TasksController and adding the complete action that would mark the task as completed:

```
class TasksController < ApplicationController
  layout false
  def index
    @tasks = Task.order('created_at DESC')
    @task = Task.new
  end

  def create
    @task = Task.new(task_params)
    @task.save
  end

  def complete
    @task = Task.find(params[:id])
    @task.update(completed: true)
  end

  private

  def task_params
    params.require(:task).permit(:title)
  end
end
```

The next step, as usual, is to update the routes:

```
Rails.application.routes.draw do
  root to: 'home#index'
  resources :tasks, only: %i[index create] do
    member do
      get 'complete'
    end
  end
end
```

The last thing we have to do is to add the complete link to the task partial that is rendered on the list. To do that, open app/views/tasks/_task.html.erb view and put there the following change:

```
<li>
  <% if task.completed %>
```

```
    <s><%= task.title %></s>
  <% else %>
    <%= task.title %> - <%= link_to('Complete',
complete_task_path(task), remote: true, data: { action:
'ajax:success->tasks#load'}) %>
  <% end %>
</li>
```

Like in the form, we can make the link execution asynchronous and listen for the ajax call result. You can now test the application and adding a new task, and marking it as a completed work without the problem; the list is always up to date.

Making updates in real-time for other users

I promised at the beginning that the whole process of adding and marking tasks as completed would be performed in real-time, and other users using the application would be aware of that. Let's make it happen.

The first step is to create the ActionCable channel through which we would send the information over web sockets:

```
touch app/channels/tasks_channel.rb
```

The channel definition is straightforward:

```
class TasksChannel < ApplicationCable::Channel
  def subscribed
    stream_from 'tasks_channel'
  end

  def unsubscribed
    stop_all_streams
  end
end
```

The next step is to update TasksController to broadcast the information when the task is created successfully or marked as completed:

```ruby
def create
  @task = Task.new(task_params)
  return unless @task.save

  ActionCable.server.broadcast('tasks_channel', {})
end

def complete
  @task = Task.find(params[:id])
  @task.update(completed: true)

  ActionCable.server.broadcast('tasks_channel', {})
end
```

The update is now broadcasted over the web sockets, but our JavaScript controller is not yet aware of that action. Open app/javascript/controllers/tasks_controller.js and put there the following code:

```javascript
import { Controller } from "stimulus"
import consumer from '../channels/consumer'

export default class extends Controller {
  static targets = ['listing'];
  static values = { url: String }

  initialize() {
    this.load();
  }

  connect() {
    this.subscription = consumer.subscriptions.create(
      {
        channel: "TasksChannel"
      },
      {
        received: this._received.bind(this),
      }
    );
 }

 _received(data) {
  this.load();
 }

  load() {
    fetch(this.urlValue)
      .then(response => response.text())
      .then(html => this.listingTarget.innerHTML = html)
  }
}
```

I did here three things:

- imported the consumer library at the top of the file
- created the web socket subscription in the connect action and hooked up the _received action that would be triggered each time the data over WebSocket will be sent
- made the definition of _received action that loads the tasks list again each time someone else updated it

You can now open the app in two separate browser windows and check that the update made in one window is automatically visible in the second window.

The next steps

It was a quick and painless development of the interactive application that updates the data in real-time for all users. What are the next steps? I encourage you to do the following:

- Build and experiment as much as you can. It's the best way to learn how to create applications efficiently
- Follow Long Live Ruby on twitter, where a secret Rails ebook is waiting for you. Over 1,100 developers already read it!
- Check out the latest book about mastering CSV in Ruby
- Wait for the next e-mails as you would receive the next portion of Ruby's awesomeness!

See you later!