# Working with JavaScript in Rails

This guide covers the options for integrating JavaScript functionality into your Rails application, including the options you have for using external JavaScript packages and how to use Turbo with Rails.

After reading this guide, you will know:

- How to use Rails without the need for a Node.js, Yarn, or a JavaScript bundler.
- How to create a new Rails application using import maps, esbuild, rollup, or webpack to bundle your JavaScript.
- What Turbo is, and how to use it.
- How to use the Turbo HTML helpers provided by Rails.

---

## Import maps

Import maps let you import JavaScript modules using logical names that map to versioned files directly from the browser. Import maps are the default from Rails 7, allowing anyone to build modern JavaScript applications using most NPM packages without the need for transpiling or bundling.

Applications using import maps do not need Node.js or Yarn to function. If you plan to use Rails with `importmap-rails` to manage your JavaScript dependencies, there is no need to install Node.js or Yarn.

When using import maps, no separate build process is required, just start your server with `bin/rails server` and you are good to go.

### Adding NPM Packages with importmap-rails

To add new packages to your import map-powered application, run the `bin/importmap pin` command from your terminal:

```
$ bin/importmap pin react react-dom
```

Then, import the package into `application.js` as usual:

```
import React from "react"
import ReactDOM from "react-dom"
```

## Adding NPM Packages with JavaScript Bundlers

Import maps are the default for new Rails applications, but if you prefer traditional JavaScript bundling, you can create new Rails applications with your choice of esbuild, webpack, or rollup.js.

To use a bundler instead of import maps in a new Rails application, pass the `—javascript` or `-j` option to `rails new`:

```
$ rails new my_new_app --javascript=webpack
OR
$ rails new my_new_app -j webpack
```

These bundling options each come with a simple configuration and integration with the asset pipeline via the jsbundling-rails gem.

When using a bundling option, use `bin/dev` to start the Rails server and build JavaScript for development.

### Installing Node.js and Yarn

If you are using a JavaScript bundler in your Rails application, Node.js and Yarn must be installed.

Find the installation instructions at the Node.js website and verify it's installed correctly with the following command:

```
$ node --version
```

The version of your Node.js runtime should be printed out. Make sure it's greater than `8.16.0`.

To install Yarn, follow the installation instructions at the Yarn website. Running this command should print out the Yarn version:

```
$ yarn --version
```

If it says something like `1.22.0`, Yarn has been installed correctly.

## Choosing Between Import Maps and a JavaScript Bundler

When you create a new Rails application, you will need to choose between import maps and a JavaScript bundling solution. Every application has different requirements, and you should consider your requirements carefully before choosing a JavaScript option, as migrating from one option to another may be time-consuming for large, complex applications.

Import maps are the default option because the Rails team believes in import maps' potential for reducing complexity, improving developer experience, and delivering performance gains.

For many applications, especially those that rely primarily on the Hotwire stack for their JavaScript needs, import maps will be the right option for the long term. You can read more about the reasoning behind making import maps the default in Rails 7 here.

Other applications may still need a traditional JavaScript bundler. Requirements that indicate that you should choose a traditional bundler include:

- If your code requires a transpilation step, such as JSX or TypeScript.
- If you need to use JavaScript libraries that include CSS or otherwise rely on Webpack loaders.
- If you are absolutely sure that you need tree-shaking.
- If you will install Bootstrap, Bulma, PostCSS, or Dart CSS through the cssbundling-rails gem. All options provided by this gem except Tailwind will automatically install `esbuild` for you if you do not specify a different option in `rails new`.

## Turbo

Whether you choose import maps or a traditional bundler, Rails ships with Turbo to speed up your application while dramatically reducing the amount of JavaScript that you will need to write.

Turbo lets your server deliver HTML directly as an alternative to the prevailing front-end frameworks that reduce the server-side of your Rails application to little more than a JSON API.

### Turbo Drive

Turbo Drive speeds up page loads by avoiding full-page teardowns and rebuilds on every navigation request. Turbo Drive is an improvement on and replacement for Turbolinks.

### Turbo Frames

Turbo Frames allow predefined parts of a page to be updated on request, without impacting the rest of the page's content.

You can use Turbo Frames to build in-place editing without any custom JavaScript, lazy load content, and create server-rendered, tabbed interfaces with ease.

Rails provides HTML helpers to simplify the use of Turbo Frames through the turbo-rails gem.

Using this gem, you can add a Turbo Frame to your application with the `turbo_frame_tag` helper like this:

```
<%= turbo_frame_tag dom_id(post) do %>
  <div>
     <%= link_to post.title, post_path(path) %>
  </div>
<% end %>
```

### Turbo Streams

Turbo Streams deliver page changes as fragments of HTML wrapped in self-executing `<turbo-stream>` elements. Turbo Streams allow you to broadcast changes made by other users over WebSockets and update pieces of a page after a form submission without requiring a full page load.

Rails provides HTML and server-side helpers to simplify the use of Turbo Streams through the turbo-rails gem.

Using this gem, you can render Turbo Streams from a controller action:

```
def create
  @post = Post.new(post_params)

  respond_to do |format|
    if @post.save
      format.turbo_stream
    else
      format.html { render :new, status: :unprocessable_entity }
    end
  end
end
```

Rails will automatically look for a `.turbo_stream.erb` view file and render that view when found.

Turbo Stream responses can also be rendered inline in the controller action:

```
def create
  @post = Post.new(post_params)

  respond_to do |format|
    if @post.save
      format.turbo_stream { render turbo_stream: turbo_stream.prepend('posts',
partial: 'post') }
    else
      format.html { render :new, status: :unprocessable_entity }
    end
  end
end
```

Finally, Turbo Streams can be initiated from a model or a background job using built-in helpers. These broadcasts can be used to update content via a WebSocket connection to all users, keeping page content fresh and bringing your application to life.

To broadcast a Turbo Stream from a model combine a model callback like this:

```
class Post < ApplicationRecord
  after_create_commit { broadcast_append_to('posts') }
end
```

With a WebSocket connection set up on the page that should receive the updates like this:

```
<%= turbo_stream_from "posts" %>
```

## Replacements for Rails/UJS Functionality

Rails 6 shipped with a tool called UJS that allows developers to override the method of `<a>` tags to perform non-GET requests after a hyperlink click and to add confirmation dialogs before executing an action. This was the default before Rails 7, but it is now recommended to use Turbo instead.

### Method

Clicking links always results in an HTTP GET request. If your application is RESTful, some links are in fact actions that change data on the server, and should be performed with non-GET requests. This attribute allows marking up such links with an explicit method such as "post", "put", or "delete".

Turbo will scan `<a>` tags in your application for the `turbo-method` data attribute and use the specified method when present, overriding the default GET action.

For example:

```
<%= link_to "Delete post", post_path(post), data: { turbo_method: "delete" } %>
```

This generates:

```
<a data-turbo-method="delete" href="...">Delete post</a>
```

An alternative to changing the method of a link with `data-turbo-method` is to use Rails `button_to` helper. For accessibility reasons, actual buttons and forms are preferable for any non-GET action.

## Confirmations

You can ask for an extra confirmation of the user by adding a `data-turbo-confirm` attribute on links and forms. The user will be presented with a JavaScript `confirm()` dialog containing the attribute's text. If the user chooses to cancel, the action doesn't take place.

Adding this attribute on links will trigger the dialog on click, and adding it on forms will trigger it on submit. For example:

```
<%= link_to "Delete post", post_path(post), data: { turbo_method: "delete",
turbo_confirm: "Are you sure?" } %>
```

This generates:

```
<a href="..." data-confirm="Are you sure?" data-turbo-method="delete">Delete
post</a>
```