Hotrails

← Back to the list of chapters

# Flash messages with Hotwire

*Published on February 24, 2022*

In this chapter, we will learn how to add flash messages with Turbo and how to make a nice animation with Stimulus.

**Sponsor this project on Github!**

This tutorial is open-source forever. If you want to support my work, you can sponsor it on Github! **I will invite you to a repository with the tutorial's source code**.

Become a sponsor

## Adding flash messages to our CRUD controller

Now that we have a working CRUD controller for our `Quote` model, we want to add flash messages to improve the usability of our application. In this chapter, we will see how flash messages work with Turbo.

Before adding flash messages *with* Turbo, we need to make them work *without* Turbo, as we used to do before Ruby on Rails 7. To do this, we will disable Turbo on the whole application, as we learned in the Turbo Drive chapter:

```
// app/javascript/application.js

import "./controllers"

// The two following lines disable Turbo on the whole application
import { Turbo } from "@hotwired/turbo-rails"
Turbo.session.drive = false
```

Now that Turbo is disabled on the whole application, we can test in the browser that our quote editor behaves like we would expect it to without Turbo. Each

**link click opens a new page**, and **each form submission redirects** to the `Quotes#index` or the `Quotes#show` page.

With Turbo disabled, it's time to start working on our flash messages. We already set flash messages in the first chapter when the `#create`, `#update`, and `#destroy` actions are successful in our `QuotesController` thanks to the `notice` option:

```
class QuotesController < ApplicationController
  # All the previous code

  def create
    @quote = current_company.quotes.build(quote_params)

    if @quote.save
      respond_to do |format|
        format.html { redirect_to quotes_path, notice: "Quote was succes
        format.turbo_stream
      end
    else
      render :new, status: :unprocessable_entity
    end
  end

  # All the previous code

  def update
    if @quote.update(quote_params)
      redirect_to quotes_path, notice: "Quote was successfully updated."
    else
      render :edit, status: :unprocessable_entity
    end
  end

  # All the previous code

  def destroy
    @quote.destroy

    respond_to do |format|
      format.html { redirect_to quotes_path, notice: "Quote was successf
      format.turbo_stream
    end
  end
end
```

---

**Note**: If you're not familiar with the `notice` notation for flash messages, the two following syntaxes are equivalent:

```
# Syntax 1
redirect_to quotes_path, notice: "Quote was successfully created."

# Syntax 2
flash[:notice] = "Quote was successfully created."
redirect_to quotes_path
```

I prefer using the first syntax as it is a one-liner, but feel free to use the second one if you want!

---

Even if flash messages are set appropriately, **we currently don't display them in the views**. Let's make sure we display those flash messages correctly for the HTML format before talking about Turbo. To do this, let's first create the flash message partial that will contain the markup for a single flash message:

```
<%# app/views/layouts/_flash.html.erb %>

<% flash.each do |flash_type, message| %>
  <div class="flash__message">
    <%= message %>
  </div>
<% end %>
```

We will render this flash message partial on every page of the application directly in the layout:

```
<!DOCTYPE html>
<html>
  <head>
    <!-- All the head code -->
  </head>

  <body>
```

```
    <%= render "layouts/navbar" %>

    <div class="flash">
      <%= render "layouts/flash" %>
    </div>

    <%= yield %>
  </body>
</html>
```

Let's test that everything is wired correctly by creating, updating, or destroying a quote. The flash message appears as expected! Let's add a little bit of CSS to make them nicer:

```
// app/assets/stylesheets/components/_flash.scss

.flash {
  position:fixed;
  top: 5rem;
  left: 50%;
  transform: translateX(-50%);

  display: flex;
  flex-direction: column;
  align-items: center;
  gap: var(--space-s);

  max-width: 100%;
  width: max-content;
  padding: 0 var(--space-m);

  &__message {
    font-size: var(--font-size-s);
    color: var(--color-white);
    padding: var(--space-xs) var(--space-m);
    background-color: var(--color-dark);
    animation: appear-then-fade 4s both;
    border-radius: 999px;
  }
}
```

The `.flash` CSS class is the container for our flash messages. It has a fixed position on the screen. Each individual flash message is then styled thanks to the `.flash__message` CSS class. We will use a custom animation for our flash messages called `appear-then-fade`. Let's add a file for animations in our CSS architecture:

```
// app/assets/stylesheets/config/_animations.scss

@keyframes appear-then-fade {
  0%, 100% {
    opacity:0
  }
  5%, 60% {
    opacity:1
  }
}
```

With those two files added, let's add them to our manifest file for them to be part of the CSS bundle:

```
// app/assets/stylesheets/application.sass.scss

@import "components/flash";
@import "config/animations";
```

Now that we added our CSS, let's test the display of our flash messages in the browser. Our flash messages are now styled appropriately!

However, there is one small glitch with our current implementation. When we hover the mouse on the flash message area, **our mouse cursor changes even when the flash message is not visible anymore**. This is because even if our flash message has an opacity of zero, it is still present in the DOM and above the rest of the page's content. **To solve this problem, we need to remove the flash messages from the DOM when they reach an opacity of zero**.

This is where we will add the **single line of JavaScript we need in the whole tutorial**. We will create a small Stimulus Controller that removes the flash message when the `appear-then-fade` animation ends.

To do this, let's type the command to generate a new Stimulus Controller called `removals`:

```
bin/rails generate stimulus removals
```

This adds a new Stimulus Controller that is imported automatically in the `app/javascript/controllers/index.js` file:

```
// app/javascript/controllers/index.js

import { application } from "./application"

import HelloController from "./hello_controller.js"
application.register("hello", HelloController)

import RemovalsController from "./removals_controller.js"
application.register("removals", RemovalsController)
```

As we can see, we have a `HelloController` that was generated automatically when we created our application at the beginning of the tutorial with the `bin/rails new` command. Let's remove it as we won't need it:

```
bin/rails destroy stimulus hello
```

This command should remove our `HelloController` and update the controllers' index file:

```
// app/javascript/controllers/index.js

import { application } from "./application"

import RemovalsController from "./removals_controller.js"
application.register("removals", RemovalsController)
```

---

**Note**: There seems to be a small bug in Rails where the two lines that register the `HelloController` are not properly removed from the controllers' index file when running the `bin/rails destroy stimulus hello` command. If that's the case for you, you can simply remove them by hand or run the `bin/rails stimulus:manifest:update` command.

---

Let's now implement our Stimulus Controller:

```
// app/javascript/controllers/removals_controller.js
```

```
import { Controller } from "@hotwired/stimulus"

export default class extends Controller {
  remove() {
    this.element.remove()
  }
}
```

This controller has a simple function called `remove`. When we call this function, it removes the DOM node where the controller is attached.

If that's a bit abstract for now, let's demonstrate how it works by using our controller on our flash messages to remove them from the DOM when their animation ends:

```
<%# app/views/layouts/_flash.html.erb %>

<% flash.each do |flash_type, message| %>
  <div
    class="flash__message"
    data-controller="removals"
    data-action="animationend->removals#remove"
  >
    <%= message %>
  </div>
<% end %>
```

**The Stimulus library allows us to link JavaScript behavior defined in Stimulus Controllers to HTML thanks to naming conventions on data attributes**.

The HTML snippet above suggests that each flash message is connected to a `RemovalsController` thanks to the `data-controller="removals"` data attribute. When the animation ends, the function `remove` of the `RemovalsController` is called thanks to the `data-action="animationend->removals#remove` data attribute.

If we test in the browser and create, update or destroy a quote, we should see the flash message appear on the page. When the animation ends, the flash message is removed from the DOM. If we inspect the DOM after a few seconds, the flash message is gone! If we hover the mouse on the flash message area, the cursor does not change anymore as the flash message was completely removed from the DOM.

Excellent! Now that everything is wired up for an application that doesn't use Turbo, it's time to ensure we have the same behavior with Turbo enabled.

## Flash messages with Turbo in Rails 7

First of all, let's **remove** the lines we added at the beginning of the chapter to disable Turbo in the whole application:

```
// app/javascript/application.js

import "./controllers"
import "@hotwired/turbo-rails"
```

Let's test in the browser. We can notice our Turbo behavior is back, but our flash messages have disappeared. Let's explain what happens.

### Flash messages with Hotwire on the `#create` action

Let's have a look at the `QuotesController#create` action. For the HTML format, the flash message is set thanks to the `notice` option. However, there is no mention of a flash message for the Turbo Stream format:

```
# app/controllers/quotes_controller.rb

def create
  @quote = current_company.quotes.build(quote_params)

  if @quote.save
    respond_to do |format|
      format.html { redirect_to quotes_path, notice: "Quote was successf
      format.turbo_stream
    end
  else
    render :new
  end
end
```

Let's add the same flash message for the Turbo Stream format:

```
# app/controllers/quotes_controller.rb

def create
  @quote = current_company.quotes.build(quote_params)

  if @quote.save
    respond_to do |format|
      format.html { redirect_to quotes_path, notice: "Quote was successf
      format.turbo_stream { flash.now[:notice] = "Quote was successfully
    end
  else
    render :new
  end
end
```

We use `flash.now[:notice]` here and not `flash[:notice]` because **Turbo Stream responses don't redirect to other locations**, so the flash has to appear on the page right now.

If we now test in the browser, the flash message still doesn't appear on the page. This is because there is no mention of what to do with the flash message in the template that gets rendered when a quote is successfully created:

```
<%# app/views/quotes/create.turbo_stream.erb %>

<%= turbo_stream.prepend "quotes", @quote %>
<%= turbo_stream.update Quote.new, "" %>
```

For our flash message to work with Turbo Stream responses, we need to add a line to instruct Turbo to *prepend* the flash messages to a list or *update* the content of the flash message container.

The Turbo Stream *action* we use depends on the effect we want to have. If we're going to stack flash messages and have a single-page application effect, we can use *prepend*. If we're going to have a single flash message on the screen at the time, we can use *replace*.

Let's use the *prepend* action in our tutorial:

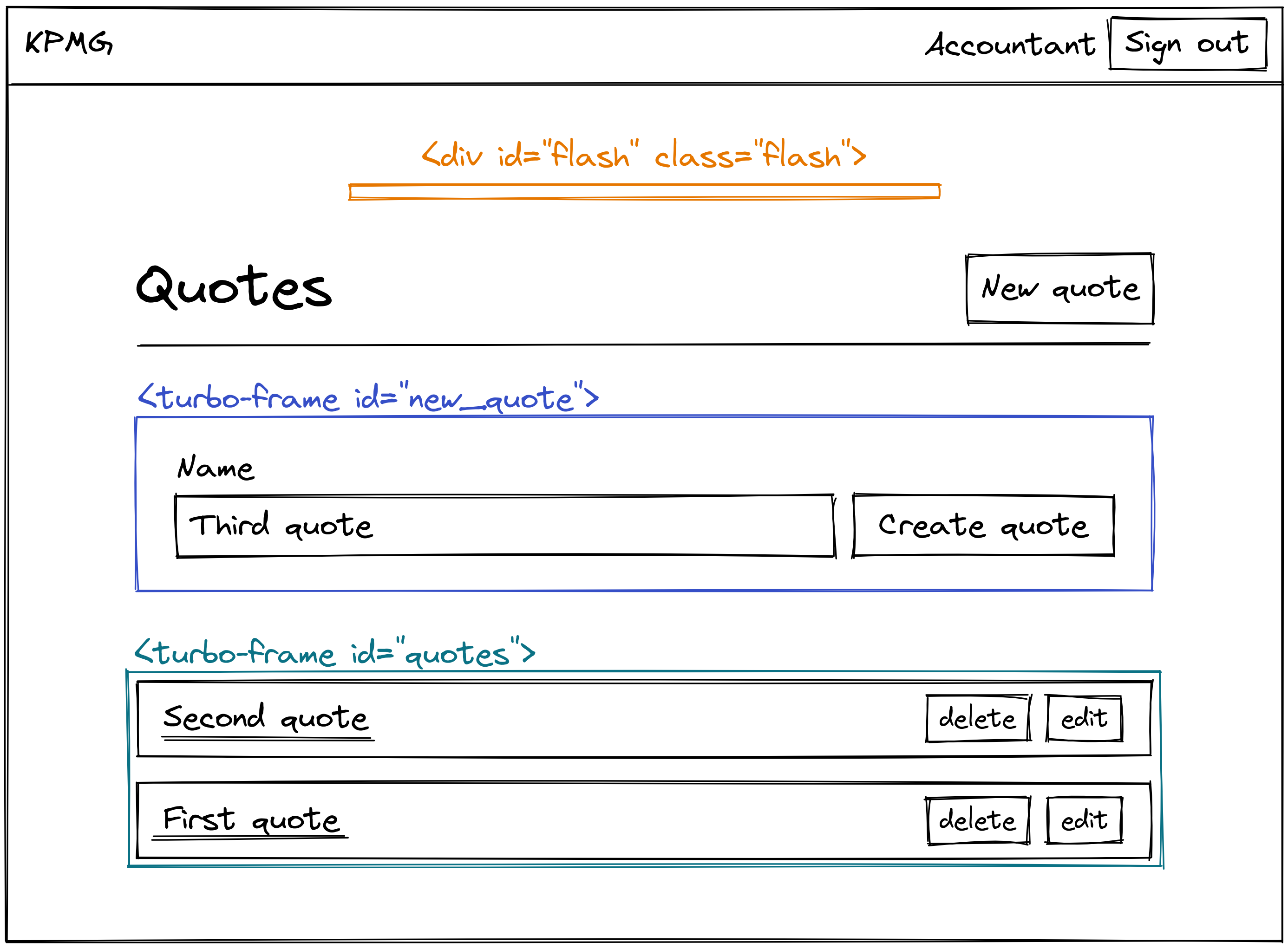
```erb
<%# app/views/quotes/create.turbo_stream.erb %>

<%= turbo_stream.prepend "quotes", @quote %>
<%= turbo_stream.update Quote.new, "" %>
<%= turbo_stream.prepend "flash", partial: "layouts/flash" %>
```

This last line instructs Turbo to prepend to the DOM node of id `flash` the content of the `layouts/flash` partial. We currently don't have a DOM node of id `flash`, so we need to add it to the application's layout:

```erb
<!DOCTYPE html>
<html>
  <head>
    <!-- All the head code -->
  </head>

  <body>
    <%= render "layouts/navbar" %>

    <div id="flash" class="flash">
      <%= render "layouts/flash" %>
    </div>

    <%= yield %>
  </body>
</html>
```

Let's test in the browser and create **two quotes very quickly**. We should see the **two flash messages appear on the screen and disappear when the animation ends**!

Let's draw a quick sketch of what happens. When we are about to create a quote, our page looks like this:

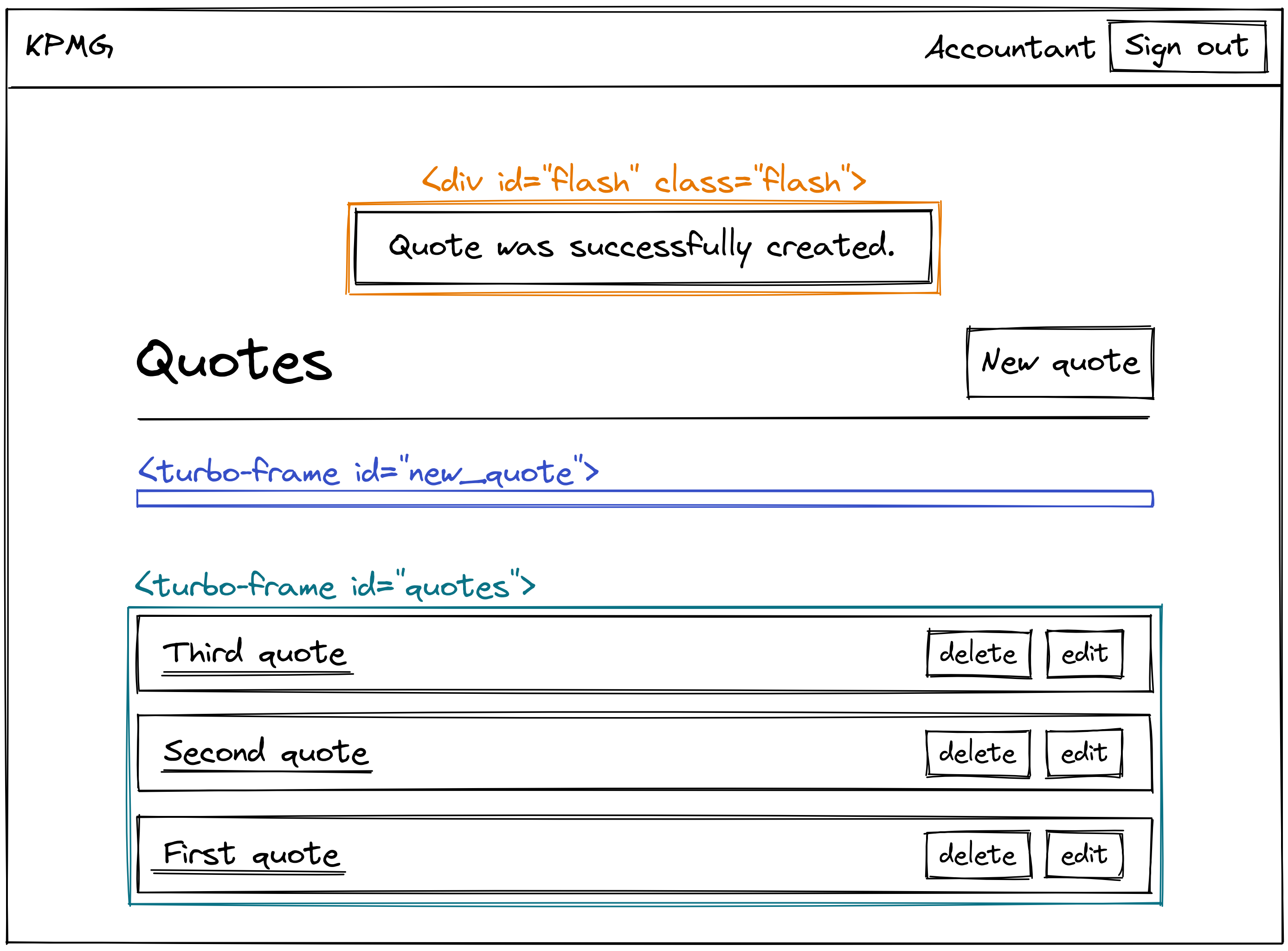


*Sketch of the Quotes#index page when we are about to create a quote*

When we submit the form, the quote is created, and the `create.turbo_stream.erb` view is rendered. This view instructs Turbo to perform three actions:

- Prepend the created quote to the DOM node with id `#quotes`
- Empty the content of the DOM node with id `#new_quote`
- Prepend the content of the flash message partial to the DOM node with id `#flash`

When those three actions are executed, our final view looks like this:


KPMG
Accountant
Sign out
<div id="flash" class="flash">
Quote was successfully created.
Quotes
New quote
<turbo-frame id="new_quote">
<turbo-frame id="quotes">
Third quote
delete
edit
Second quote
delete
edit
First quote
delete
edit

*Sketch of the Quotes#index page when we just created a quote*

Now that our flash message work for the `QuotesController#create` action, let's add flash messages to our `QuotesController#update` and `QuotesController#destroy` actions. Feel free to try to do it by yourself before reading the following two sections!

## Flash messages with Hotwire on the `#update` action

Unlike the `#create` action, the `#update` action does not have a specific view for Turbo Stream responses. If we test in our browser, the flash message does not appear on the page.

As discussed in Chapter 4, because the quote edition form is inside a Turbo Frame, this Turbo Frame is isolated from the rest of the page. When updating a quote, even if the response contains the flash message, **Turbo will only extract and replace the Turbo Frame corresponding to the quote that was just updated**.

If we want to add flash messages to our `#update` action, we have to create a Turbo Stream view just like we did with the `#create` action. The first thing we need to do is to change the `#update` method in the controller:

```
def update
  if @quote.update(quote_params)
    respond_to do |format|
      format.html { redirect_to quotes_path, notice: "Quote was successf
      format.turbo_stream { flash.now[:notice] = "Quote was successfully
    end
  else
    render :edit, status: :unprocessable_entity
  end
end
```

Now that our controller supports the Turbo Stream format, we have to create a Turbo Stream view:

```
<%# app/views/quotes/update.turbo_stream.erb %>

<%= turbo_stream.replace @quote %>
<%= turbo_stream.prepend "flash", partial: "layouts/flash" %>
```

Let's test it in our browser. As we can see, the quote is replaced when updated, and the flash message is rendered as expected!

## Flash messages with Hotwire on the `#destroy` action

The code we will write for the `#destroy` action is similar to what we just wrote for the `#create`, and `#update` actions. Let's first add the flash message we want to display when the controller responds to a Turbo Stream request:

```
def destroy
  @quote.destroy

  respond_to do |format|
    format.html { redirect_to quotes_path, notice: "Quote was successful
    format.turbo_stream { flash.now[:notice] = "Quote was successfully d
  end
end
```

Just like for the `#create` and `#update` actions, we also have to *prepend* our new flash messages to the list of flash messages:

```
<%# app/views/quotes/destroy.turbo_stream.erb %>

<%= turbo_stream.remove @quote %>
<%= turbo_stream.prepend "flash", partial: "layouts/flash" %>
```

Let's test it in our browser, and it works as expected!

## Refactoring our flash messages with a helper

In the three Turbo Stream views we just created, we use the same line everywhere to render flash messages:

```
<%= turbo_stream.prepend "flash", partial: "layouts/flash" %>
```

This line of code is repeated in three of our views already and has several **reasons to change**. For example, we could decide to use *update* instead of *prepend*, or change the path of the `layouts/flash` partial to `components/flash`.

To get immune to those changes, one strategy is to remove the duplication in our code. Removing duplication is commonly referred to as the DRY principle (Don't Repeat Yourself). That way, if `layouts/flash` later changes to `components/flash`, we only have a single place to change it.

In our example, we will remove duplication by creating a helper. We will define the method in the `ApplicationHelper`:

```
# app/helpers/application_helper.rb

module ApplicationHelper
  def render_turbo_stream_flash_messages
    turbo_stream.prepend "flash", partial: "layouts/flash"
  end
end
```

We can now use that helper in our three Turbo Stream views:

```
<%# app/views/quotes/create.turbo_stream.erb %>

<%= turbo_stream.prepend "quotes", @quote %>
<%= turbo_stream.update Quote.new, "" %>
<%= render_turbo_stream_flash_messages %>
```

```
<%# app/views/quotes/update.turbo_stream.erb %>

<%= turbo_stream.replace @quote %>
<%= render_turbo_stream_flash_messages %>
```

```
<%# app/views/quotes/destroy.turbo_stream.erb %>

<%= turbo_stream.remove @quote %>
<%= render_turbo_stream_flash_messages %>
```

With this helper, we can now safely change how our flash messages behave in our whole application. Our code now looks very clean, and we are done with flash messages!

## Wrap up

Flash messages are an important tool to give more information to users.

Making flash messages work with Hotwire in Rails 7 requires a little more setup than in previous versions of Rails, but we can now easily add nice effects to them. For example, it is possible to stack flash messages when multiple operations happen in a short time frame!

In the next chapter, we will see another very important tool for user experience in our applications: empty states! See you there!

← previous next →

Get notified when I write new articles

If you liked this article and want to keep up with Ruby on Rails and Hotwire, you can subscribe to my newsletter (no spam, no tracking, unsubscribe any time)!

Subscribe to the newsletter

Github Twitter Newsletter

*Made with remotely*