Hotrails

← Back to the list of chapters

Nested Turbo Frames

Published on March 14, 2022

In this chapter, we will build our last CRUD controller for line items. As line items are nested in line item dates, we will have some interesting challenges to solve with Turbo Frames!

What we will build in this chapter

We will *almost* finalize our quote editor in this chapter by adding **line items** to the **line item dates** we created in the previous chapter. Those line items will have a `name`, an optional `description`, a `unit_price`, and a `quantity`.

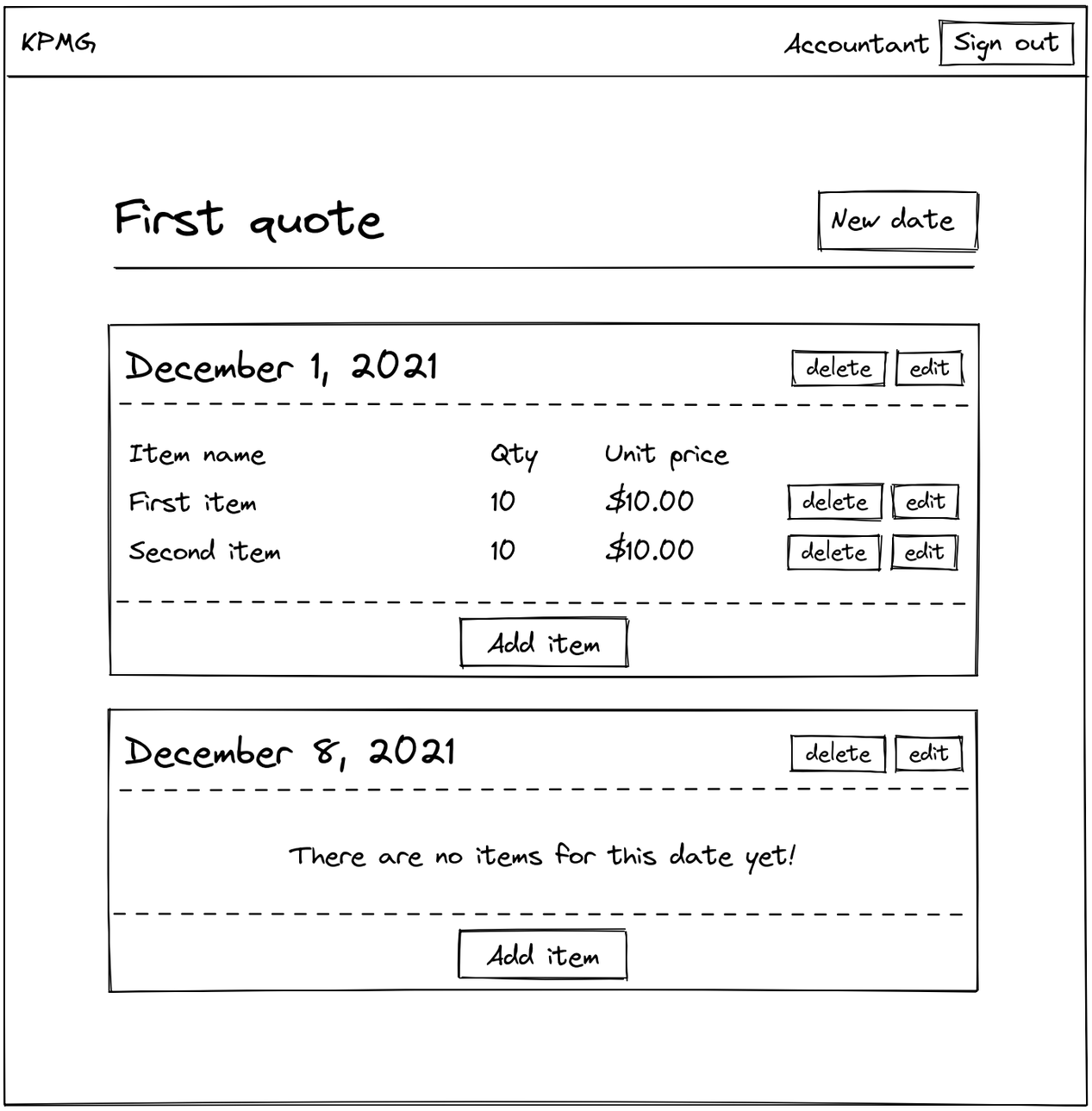
While this chapter is about another CRUD controller, it will come with some interesting challenges as we will have a lot of nested Turbo Frames. We will also discuss how to preserve the *state* of our `Quotes#show` page when performing CRUD operations on both the `LineItemDate` and the `LineItem` models.

Before we start coding, let's look at how our final quote editor will work. Let's create a quote and then navigate to the `Quotes#show` page. Let's then create a few **line item dates** and a few **line items** to have a solid grasp of what our final product should look like.

Once we have a good understanding of how the final quote editor behaves, let's start, as always, by making our CRUD controller on the `LineItem` model work

without Turbo Frames and Turbo Streams. We will add Turbo Rails features later once our controller works properly.

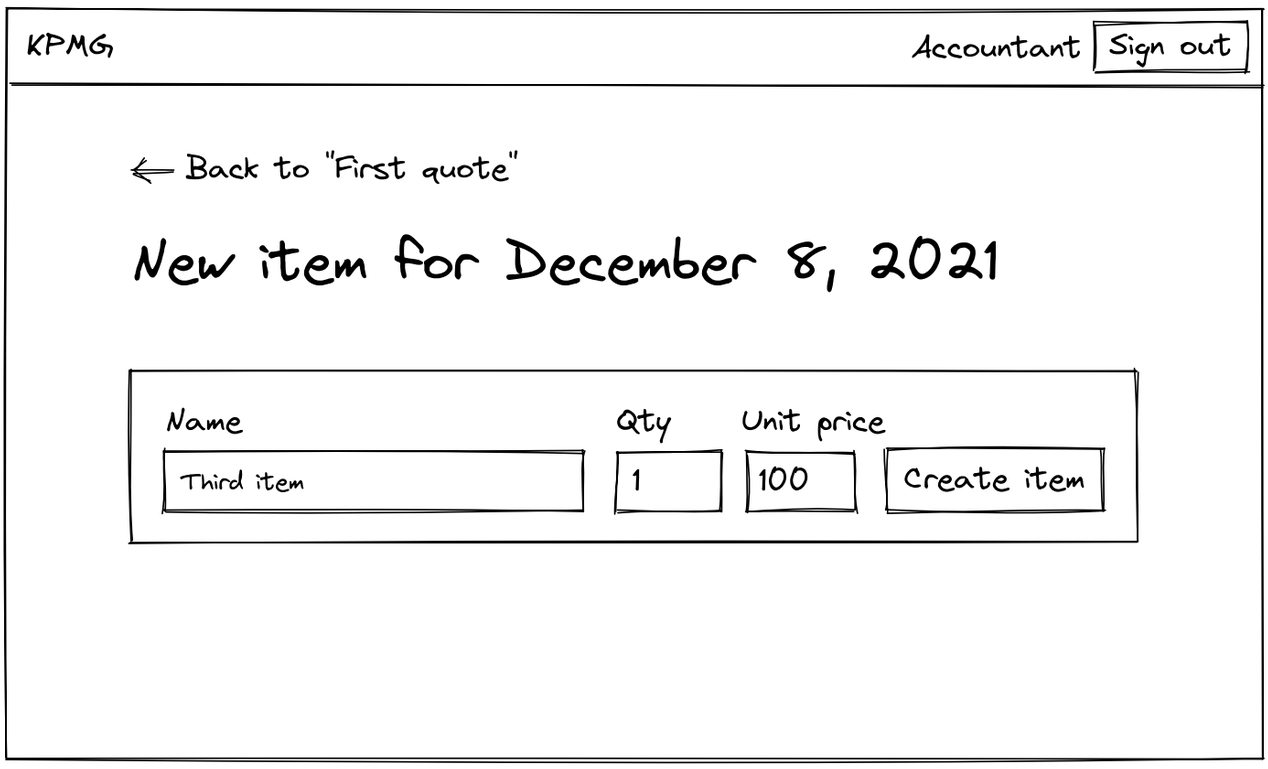
First, let's make a few sketches of how our application will behave **without Turbo Frames and Turbo Streams**. When we visit the `Quotes#show` page, we now have **line item dates** on our quote. As each **line item date** will be able to have multiple **line items**, each **line item date** card will have a "Add item" link to add **line items** for this specific **line item date**:



Sketch of the Quotes#show page with some dates and some items

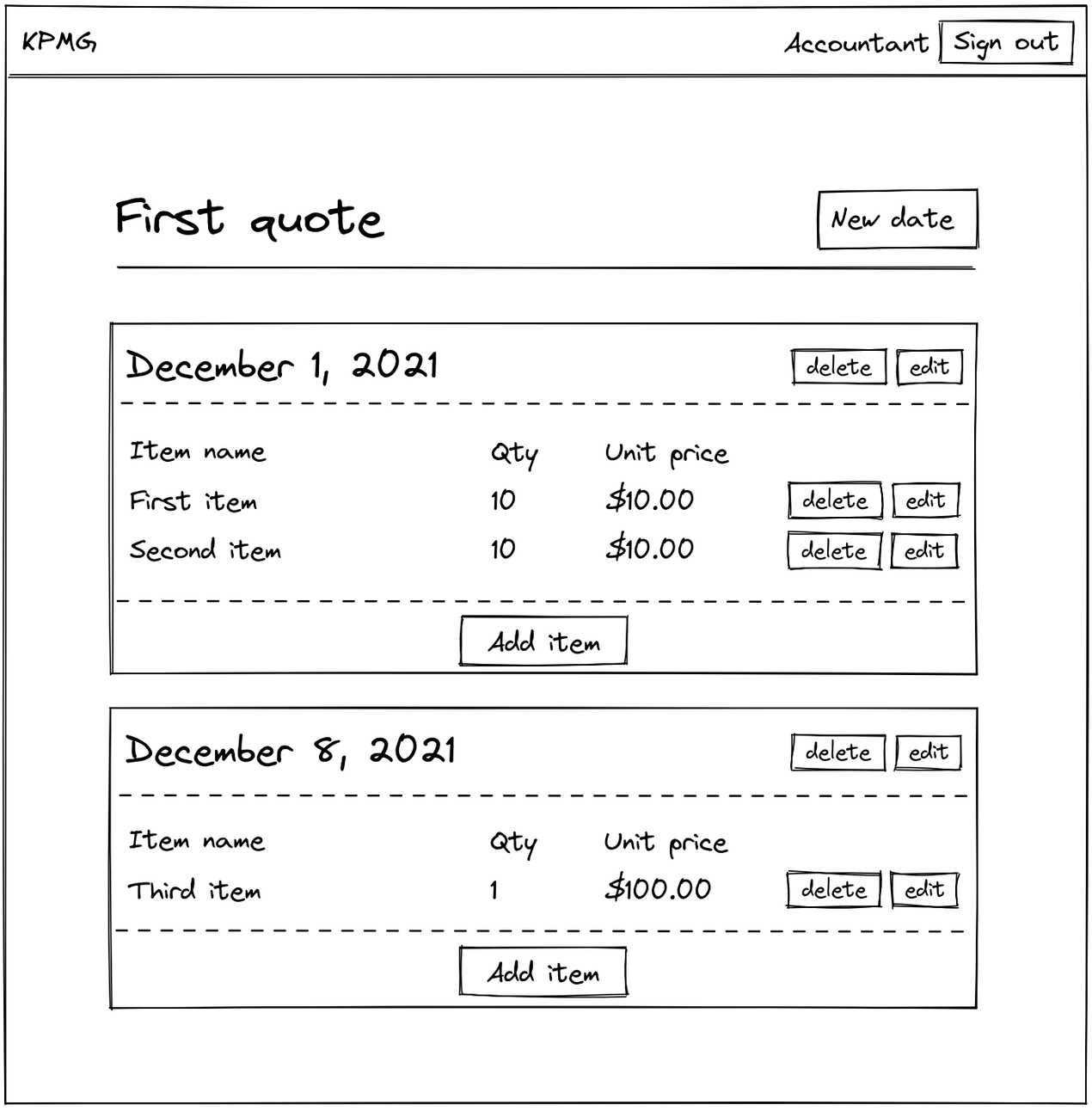
On the `Quotes#show` page, we should be able to add **line items** to any of the **line item dates** present on the quote. As we are first building the CRUD **without Turbo**, clicking on the "Add item" link for a **line item date** will take us to the `LineItems#new` page, where we can add a **line item** for this specific **line item date**.

If we click on the "Add item" link for the **second line item date**, here is a sketch of the page we can expect:



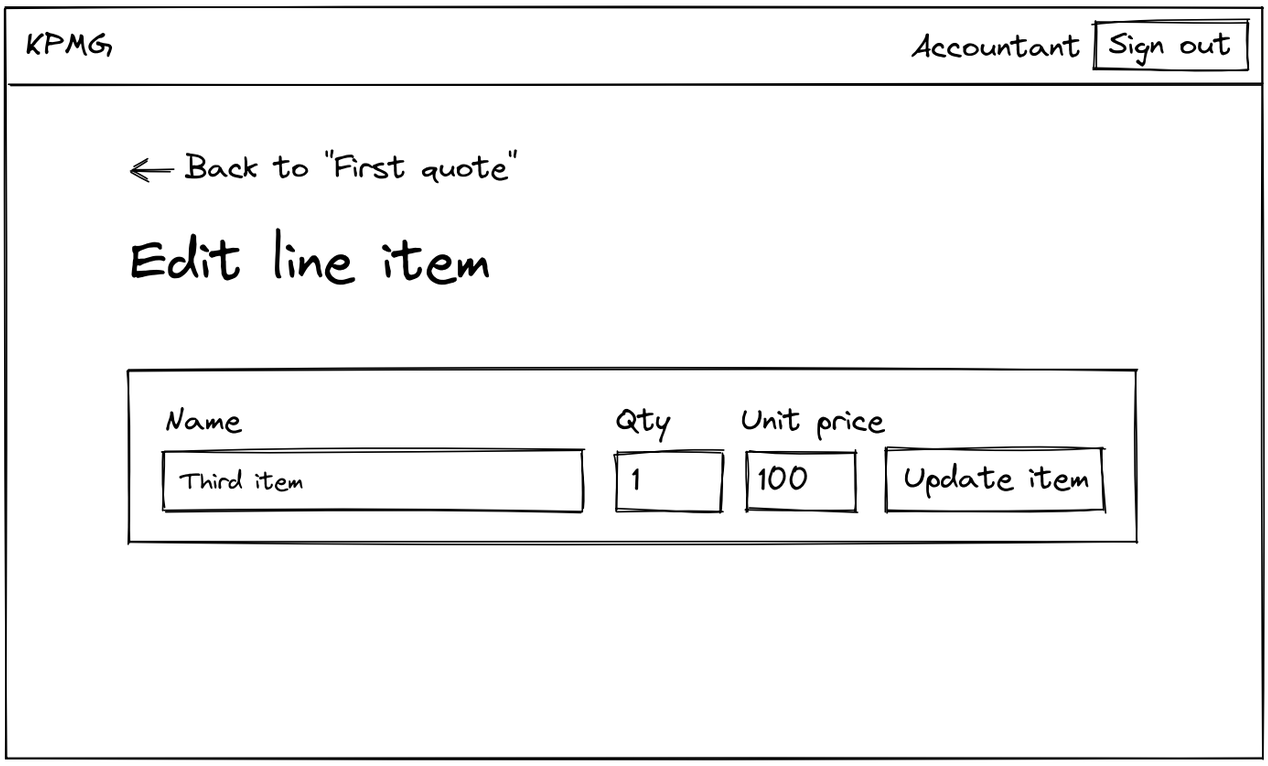
Sketch of the LineItems#new page for the second date

If we submit a valid form, we will be redirected to the `Quotes#show` page with the new **line item** added:



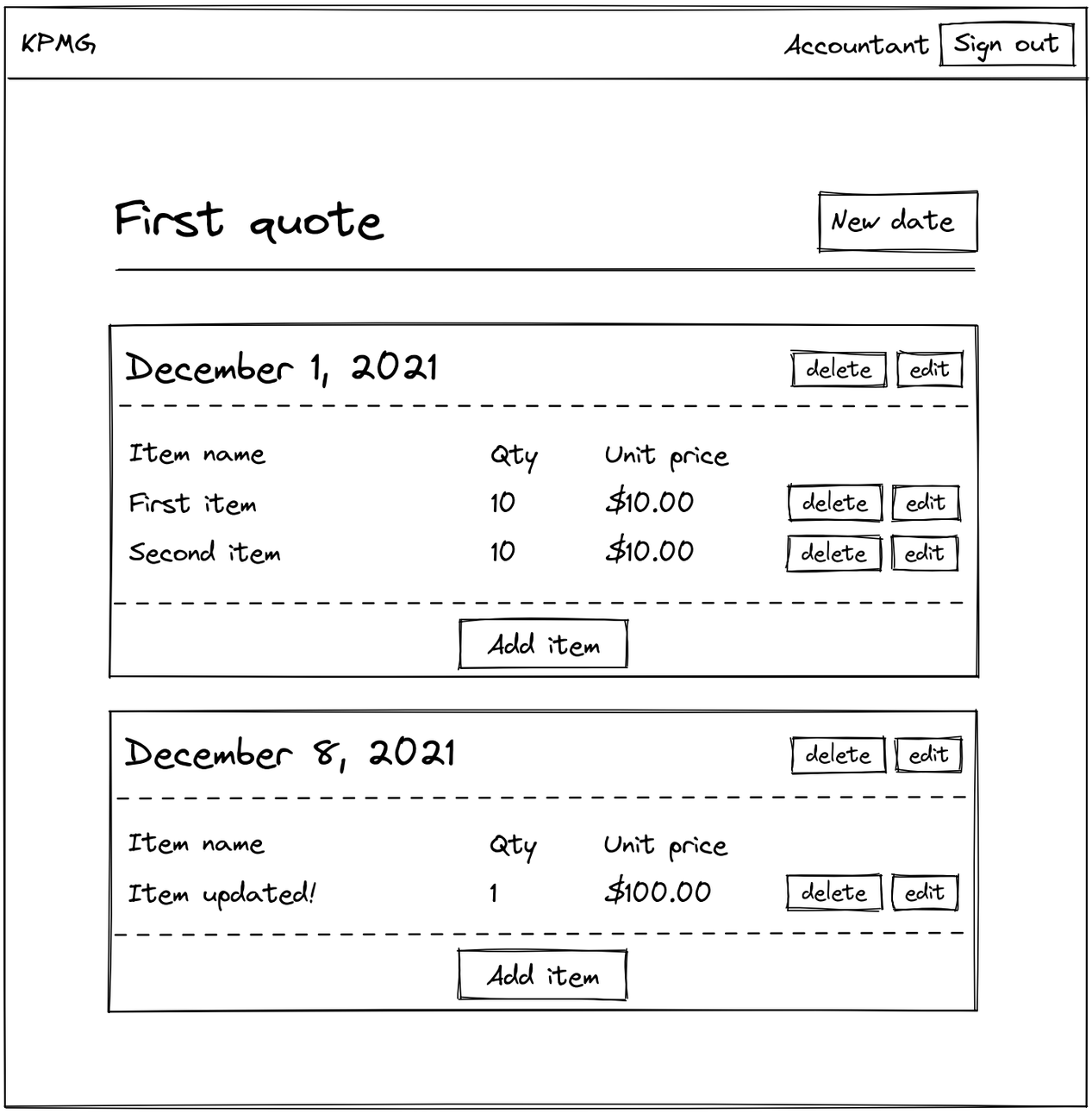
Sketch of the Quotes#show page with the created line item added to the list

If we decide to update the **line item** we just created, we can click on the "Edit" link for this **line item** to navigate to the `LineItems#edit` page:



Sketch of the LineItems#edit page

If we submit a valid form, we will be redirected to the `Quotes#show` page with the **line item** updated:



Sketch of the Quotes#show page with the updated line item

Last but not least, we can delete a **line item** by clicking on the "Delete" link for this **line item**. The **line item** is then removed from the list.

Now that the requirements are clear, it's time to start coding!

Creating the model

Let's start by creating a model named `LineItem`. This model will have five fields:

- a reference to the **line item date** it belongs to
- a name
- an optional description
- a unit price
- a quantity

We add a reference to the `LineItemDate` model because each **line item** belongs to a **line item date**, and each **line item date** has many **line items**. Let's generate the migration:

```
bin/rails generate model LineItem \
  line_item_date:references \
  name:string \
  description:text \
  quantity:integer \
  unit_price:decimal{10-2}
```

Before running the `rails db:migrate` command, we must add constraints to the `name`, `quantity`, and `unit_price` fields as we want them to always be present on each record. We can enforce this validation at the database level thanks to the `null: false` constraint.

The final migration should look like this:

```
# db/migrate/XXXXXXXXXXXXXX_create_line_items.rb

class CreateLineItems < ActiveRecord::Migration[7.0]
  def change
    create_table :line_items do |t|
      t.references :line_item_date, null: false, foreign_key: true
      t.string :name, null: false
      t.text :description
      t.integer :quantity, null: false
      t.decimal :unit_price, precision: 10, scale: 2, null: false

      t.timestamps
    end
  end
end
```

Now that our migration is ready, we can run it:

```
bin/rails db:migrate
```

Let's add the associations and the corresponding validations on the `LineItem` model:

```ruby
# app/models/line_item.rb

class LineItem < ApplicationRecord
  belongs_to :line_item_date

  validates :name, presence: true
  validates :quantity, presence: true, numericality: { only_integer: tru
  validates :unit_price, presence: true, numericality: { greater_than: 0

  delegate :quote, to: :line_item_date
end
```

The validations on the model enforce that:

- The `name`, `quantity`, and `unit_price` fields must be present
- The `unit_price` and `quantity` fields must be greater than zero
- The `quantity` must be an integer

We also delegate the `quote` method to the `LineItem#line_item_date` method. That way, the two following lines are equivalent:

```ruby
line_item.line_item_date.quote
line_item.quote
```

Now that our `LineItem` model is completed, let's add the `has_many` association on the `LineItemDate` model:

```ruby
# app/models/line_item_date.rb

class LineItemDate < ApplicationRecord
  has_many :line_items, dependent: :destroy

  # All the previous code...
end
```

Our model layer is now complete! Let's next work on the routes.

Adding routes for line items

We want to perform all the seven CRUD actions on the `LineItem` model except two of them:

- We won't need the `LineItem#index` action as all line items will already be present on the `Quotes#show` page.
- We won't need the `LineItem#show` action as it would make no sense for us to view a single line item. We always want to see the quote as a whole.

Those two exceptions are reflected in the routes file below:

```
# config/routes.rb

Rails.application.routes.draw do
  # All the previous routes

  resources :quotes do
    resources :line_item_dates, except: [:index, :show] do
      resources :line_items, except: [:index, :show]
    end
  end
end
```

Now that our routes are ready, it's time to start designing our application with fake data!

Designing line items

The **line item dates** of the `Quotes#show` page currently don't have any **line items**. Let's fix that by adding some fake data to our fixtures.

Let's imagine that the quote editor we are building is for a corporate events software. As events can span multiple dates, our quotes will have multiple dates and each will have multiple items! In our fixture file, we might want to add a room where the guests can have a meeting and a meal. Let's add those items in the fixtures file:

```
# test/fixtures/line_items.yml

room_today:
  line_item_date: today
```

```
  name: Meeting room
  description: A cosy meeting room for 10 people
  quantity: 1
  unit_price: 1000

catering_today:
  line_item_date: today
  name: Meal tray
  description: Our delicious meal tray
  quantity: 10
  unit_price: 25

room_next_week:
  line_item_date: next_week
  name: Meeting room
  description: A cosy meeting room for 10 people
  quantity: 1
  unit_price: 1000

catering_next_week:
  line_item_date: next_week
  name: Meal tray
  description: Our delicious meal tray
  quantity: 10
  unit_price: 25
```

We can now seed the database again by running the `bin/rails db:seed` command. Those seeds will enable us to design our quote editor with fake data. Let's now open the application on the `Quotes#show` page for the "First quote". We want to add two elements to each **line item date** on the page:

- The collection of line items for this date
- The link to add new line items for this date

Let's add those elements by completing the partial for a single line item date:

```
<%# app/views/line_item_dates/_line_item_date.html.erb %>

<%= turbo_frame_tag line_item_date do %>
  <div class="line-item-date">
    <div class="line-item-date__header">
      <!-- All the previous code -->
    </div>
```

```
    <div class="line-item-date__body">
      <div class="line-item line-item--header">
        <div class="line-item__name">Article</div>
        <div class="line-item__quantity">Quantity</div>
        <div class="line-item__price">Price</div>
        <div class="line-item__actions"></div>
      </div>

      <%= render line_item_date.line_items, quote: quote, line_item_date

      <div class="line-item-date__footer">
        <%= link_to "Add item",
                    [:new, quote, line_item_date, :line_item],
                    class: "btn btn--primary" %>
      </div>
    </div>
  </div>
<% end %>
```

To render each line item, we now have to create a partial to display a single line item:

```
<%# app/views/line_items/_line_item.html.erb %>

<div class="line-item">
  <div class="line-item__name">
    <%= line_item.name %>
    <div class="line-item__description">
      <%= simple_format line_item.description %>
    </div>
  </div>
  <div class="line-item__quantity-price">
    <%= line_item.quantity %>
    &times;
    <%= number_to_currency line_item.unit_price %>
  </div>
  <div class="line-item__quantity">
    <%= line_item.quantity %>
  </div>
  <div class="line-item__price">
    <%= number_to_currency line_item.unit_price %>
  </div>
  <div class="line-item__actions">
```

```
    <%= button_to "Delete",
                  [quote, line_item_date, line_item],
                  method: :delete,
                  class: "btn btn--light" %>
    <%= link_to "Edit",
                [:edit, quote, line_item_date, line_item],
                class: "btn btn--light" %>
  </div>
</div>
```

The `simple_format` helper is often useful to render text that was typed into a textarea. For example, let's imagine a user typed the following text in the description field for a **line item**:

```
- Appetizer
- Main course
- Dessert
- A glass of wine
```

The HTML generated by the `simple_format` helper in our view will be:

```
<p>
  - Appetizers
  <br>
  - Main course
  <br>
  - Dessert
  <br>
  - A glass of wine
</p>
```

As we can see, the formatting our user wanted when he filled his text area is preserved thanks to the line breaks `<br>`. If we remove the `simple_format` helper, the text will be displayed on a single line which is not what we want here.

The `.line-item__quantity`, `.line-item__price`, and `.line-item__quantity-price` CSS classes might seem a bit redundant, but we will display the two first CSS classes only when the screen size is above our `tabletAndUp` breakpoint, and we will display the last CSS class only on mobile.

Now that we have the HTML markup let's add a little bit of CSS to make our design a bit nicer. The first thing we have to do is finalize our `.line-item-date` component that we started in the previous chapter by adding the `.line-item-date__body` and `.line-item-date__footer` elements:

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

.line-item-date {
  // All the previous code

  &__body {
    border-radius: var(--border-radius);
    background-color: var(--color-white);
    box-shadow: var(--shadow-small);
    margin-top: var(--space-xs);
    padding: var(--space-xxs);
    padding-top: 0;

    @include media(tabletAndUp) {
      padding: var(--space-m);
    }
  }

  &__footer {
    border: dashed 2px var(--color-light);
    border-radius: var(--border-radius);
    text-align: center;
    padding: var(--space-xxs);

    @include media(tabletAndUp) {
      padding: var(--space-m);
    }
  }
}
```

Now that our `.line-item-date` CSS component is complete let's spend some time designing each individual **line item**. We are going to write a lot of CSS here as we will create:

- Our `.line-item` base component to design a single **line item**
- A `.line-item--header` modifier to design the header row of a collection of **line items**

- A `.line-item--form` modifier to design the form to create and update a **line item**

All those components will be responsive on **mobile** and **tablets, and larger screens** thanks to our `tabletAndUp` breakpoint. Let's dive into the code:

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

.line-item {
  display: flex;
  align-items: start;
  flex-wrap: wrap;
  background-color: var(--color-white);

  gap: var(--space-xs);
  margin-bottom: var(--space-s);
  padding: var(--space-xs);
  border-radius: var(--border-radius);

  > * {
    margin-bottom: 0;
  }

  &__name {
    flex: 1 1 100%;
    font-weight: bold;

    @include media(tabletAndUp) {
      flex: 1 1 0;
    }
  }

  &__description {
    flex-basis: 100%;
    max-width: 100%;
    color: var(--color-text-muted);
    font-weight: normal;
    font-size: var(--font-size-s);
  }

  &__quantity-price {
    flex: 0 0 auto;
    align-self: flex-end;
    justify-self: flex-end;
```

```
    order: 3;

    font-weight: bold;

    @include media(tabletAndUp) {
      display: none;
    }
  }

  &__quantity {
    flex: 1;
    display: none;

    @include media(tabletAndUp) {
      display: revert;
      flex: 0 0 7rem;
    }
  }

  &__price {
    flex: 1;
    display: none;

    @include media(tabletAndUp) {
      display: revert;
      flex: 0 0 9rem;
    }
  }

  &__actions {
    display: flex;
    gap: var(--space-xs);
    order: 2;
    flex: 1 1 auto;

    @include media(tabletAndUp) {
      order: revert;
      flex: 0 0 10rem;
    }
  }

  &--form {
    box-shadow: var(--shadow-small);

    .line-item__quantity,
```

```
    .line-item__price {
      display: block;
    }

    .line-item__description {
      order: 2;
    }
  }

  &--header {
    display: none;
    background-color: var(--color-light);
    margin-bottom: var(--space-s);

    @include media(tabletAndUp) {
      display: flex;
    }

    & > * {
      font-size: var(--font-size-s);
      font-weight: bold ;
      letter-spacing: 1px;
      text-transform: uppercase;
    }
  }
}
```

Let's not forget to import this new file inside our manifest file:

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

// All the previous code
@import "components/line_item";
```

That was **a lot** of CSS, but everything should look nice now! If we open the browser, we should see that our design is *good enough*!

Before moving to the next section, let's notice that we currently have a **performance problem**. Even though it is a bit outside of the scope of this tutorial, it's important to explain what happens here. If we inspect the logs of our Rails server when visiting the `Quotes#show` page, we will see an *N+1 query* issue here:

```
...
SELECT "line_items".* FROM "line_items" WHERE "line_items"."line_item_da
...
SELECT "line_items".* FROM "line_items" WHERE "line_items"."line_item_da
...
```

In the extract from the logs above, we are querying the `line_items` table twice because we have two **line item dates**, but we would query it **n** times if we had **n line item dates**. This is because each time we render a new **line item date**, we also perform a request to retrieve the associated **line items** because of this line:

```
<%# app/views/line_item_dates/_line_item_date.html.erb %>

<%= render line_item_date.line_items, quote: quote, line_item_date: line
```

A good rule of thumb for performance is that we should query a database table only once per request-response cycle.

To avoid the *N+1 query* issue, we need to load the collection of **line items** for each **line item date** in advance. Let's do it in the `QuotesController#show` action:

```
# app/controllers/quotes_controller.rb

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

  def show
    @line_item_dates = @quote.line_item_dates.includes(:line_items).orde
  end

  # All the previous code...
end
```

With this `includes` added, we should notice in the logs that we now only query the `line_items` table **once** to display the page:

```
SELECT "line_items".* FROM "line_items" WHERE "line_items"."line_item_da
```

With that performance issue solved, it is now time to create our `LineItemsController` and make it work!

Our standard CRUD controller

Creating line items without Turbo

Now that our database schema, model, routes, markup, and design are done, it's time to start working on the controller. As mentioned in the introduction, we will first build a standard controller **without Turbo Frames and Turbo Streams; we will add them later**.

Our controller will contain all the seven actions of the CRUD except the `#index` and the `#show` actions. Let's start by making the `#new` and `#create` actions work:

```
# app/controllers/line_items_controller.rb

class LineItemsController < ApplicationController
  before_action :set_quote
  before_action :set_line_item_date

  def new
    @line_item = @line_item_date.line_items.build
  end

  def create
    @line_item = @line_item_date.line_items.build(line_item_params)

    if @line_item.save
      redirect_to quote_path(@quote), notice: "Item was successfully cre
    else
      render :new, status: :unprocessable_entity
    end
  end

  private

  def line_item_params
    params.require(:line_item).permit(:name, :description, :quantity, :u
  end
```

```
  def set_quote
    @quote = current_company.quotes.find(params[:quote_id])
  end

  def set_line_item_date
    @line_item_date = @quote.line_item_dates.find(params[:line_item_date
  end
end
```

Our controller is very standard and should already work, but we are missing the `line_items/new.html.erb` view and the `line_items/_form.html.erb` partial. Let's add those two files to our application:

```
<%# app/views/line_items/new.html.erb %>

<main class="container">
  <%= link_to sanitize("&larr; Back to quote"), quote_path(@quote) %>

  <div class="header">
    <h1>New item for <%= l(@line_item_date.date, format: :long) %></h1>
  </div>

  <%= render "form",
             quote: @quote,
             line_item_date: @line_item_date,
             line_item: @line_item %>
</main>
```

We don't need a fancy design for our `LineItems#new` page as we will use Turbo later to extract the form from the page and insert it in the `Quotes#show` page. However, it should still be usable for people using legacy browsers that don't support Turbo. Let's add the markup for our form:

```
<%# app/views/line_items/_form.html.erb %>

<%= simple_form_for [quote, line_item_date, line_item],
                    html: { class: "form line-item line-item--form" } do

  <%= form_error_notification(line_item) %>

  <%= f.input :name,
```

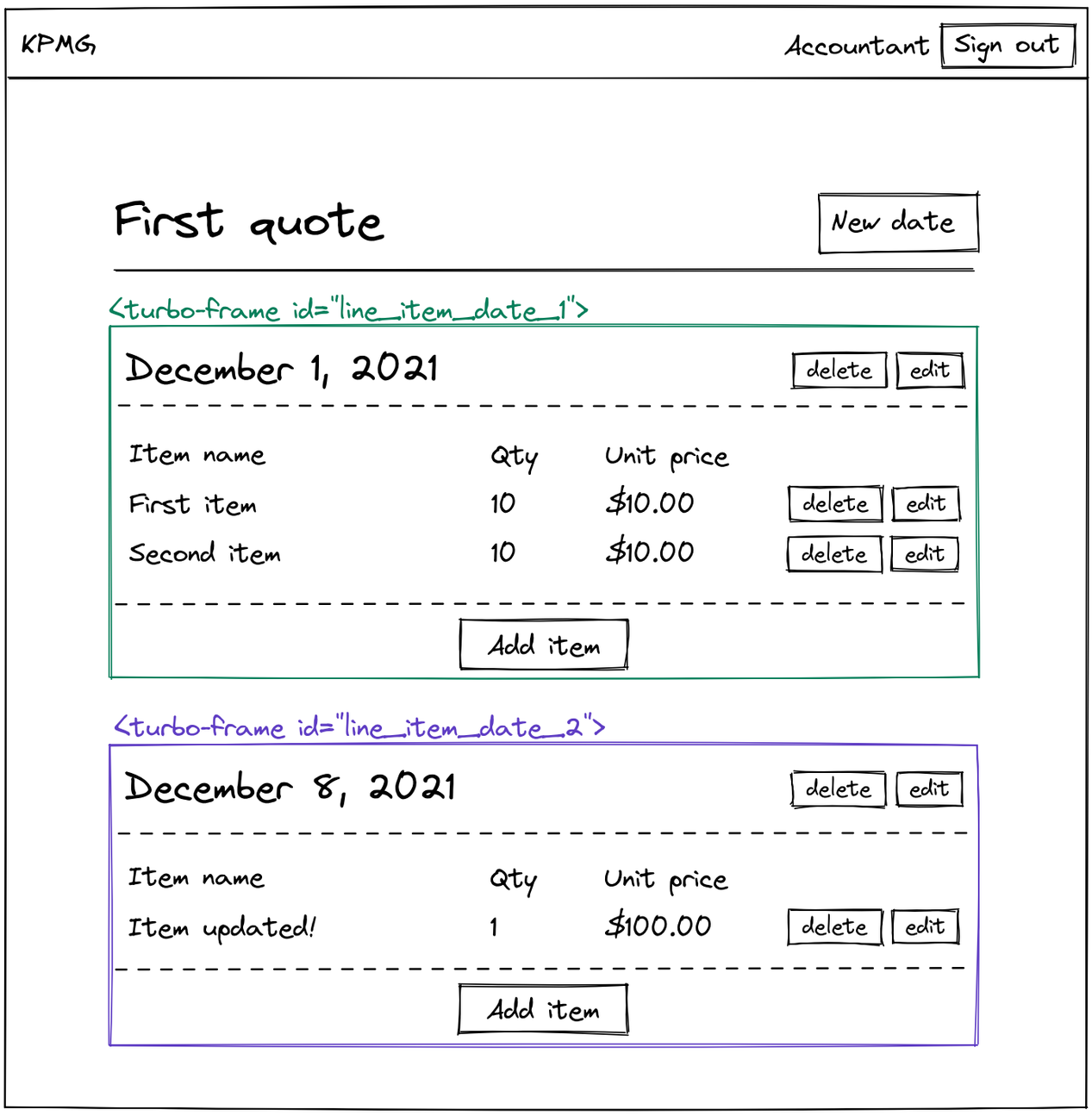
```
              wrapper_html: { class: "line-item__name" },
              input_html: { autofocus: true } %>
  <%= f.input :quantity,
              wrapper_html: { class: "line-item__quantity" } %>
  <%= f.input :unit_price,
              wrapper_html: { class: "line-item__price" } %>
  <%= f.input :description,
              wrapper_html: { class: "line-item__description" } %>

  <div class="line-item__actions">
    <%= link_to "Cancel", quote_path(quote), class: "btn btn--light" %>
    <%= f.submit class: "btn btn--secondary" %>
  </div>
<% end %>
```

In this form, we reuse the `form_error_notification` helper that we created in the last chapter! We also reuse the `.line-item` CSS class in combination with the `.line-item--form` modifier to style the form.

Let's test in our browser. **It does not work as expected**! The **line item date** card disappears, and we have the following error in the browser's console:

```
Response has no matching <turbo-frame id="line_item_date_123456"> elemen
```

This is because our "Add item" link is already nested within a Turbo Frame we added in the previous chapter as described in the sketch below:



Sketch of the Quotes#show with the Turbo Frames from the previous chapter

Because of those Turbo Frames, Turbo will intercept all clicks on links and form submissions within those Turbo Frames and expect a Turbo Frame of the same id in the response. We first want to make our CRUD work **without Turbo Frames and Turbo Streams**.

To prevent Turbo from intercepting our clicks on links and form submissions, we will use the `data-turbo-frame="_top"` data attribute as explained in the first chapter about Turbo Frames. Let's add this data attribute to our "Add item" link:

```
<%# app/views/line_item_dates/_line_item_date.html.erb %>

<!-- All the previous code -->

<div class="line-item-date__footer">
  <%= link_to "Add item",
```

```
              [:new, quote, line_item_date, :line_item],
              data: { turbo_frame: "_top" },
              class: "btn btn--primary" %>
</div>

<!-- All the previous code -->
```

Let's also anticipate and add the same data attribute to the "Edit" link and the "Delete" form on the line item partial as we will have the same issue there:

```
<%# app/views/line_items/_line_item.html.erb %>

<!-- All the previous code -->

<div class="line-item__actions">
  <%= button_to "Delete",
                [quote, line_item_date, line_item],
                method: :delete,
                form: { data: { turbo_frame: "_top" } },
                class: "btn btn--light" %>
  <%= link_to "Edit",
              [:edit, quote, line_item_date, line_item],
              data: { turbo_frame: "_top" },
              class: "btn btn--light" %>
</div>

<!-- All the previous code -->
```

Let's now test our `#new` and `#create` action in the browser. Everything now works as expected!

Let's just take a few seconds to fill the translations file with the text we want for the labels, placeholders, and submit buttons:

```
# config/locales/simple_form.en.yml

en:
  simple_form:
    placeholders:
      quote:
        name: Name of your quote
      line_item:
        name: Name of your item
```

```
        description: Description (optional)
        quantity: 1
        unit_price: $100.00
    labels:
      quote:
        name: Name
      line_item:
        name: Name
        description: Description
        quantity: Quantity
        unit_price: Unit price
      line_item_date:
        date: Date

  helpers:
    submit:
      quote:
        create: Create quote
        update: Update quote
      line_item:
        create: Create item
        update: Update item
      line_item_date:
        create: Create date
        update: Update date
```

With that file completed, the text of the submit button will be "Create date" when we create a `LineItemDate` and "Update date" when we update a `LineItemDate`.

Updating line items without Turbo

Now that our `#new` and `#create` actions are working let's do the same work for the `#edit` and `#update` actions. Let's start with the controller:

```
class LineItemsController < ApplicationController
  before_action :set_quote
  before_action :set_line_item_date
  before_action :set_line_item, only: [:edit, :update, :destroy]

  # All the previous code

  def edit
```

```
  end

  def update
    if @line_item.update(line_item_params)
      redirect_to quote_path(@quote), notice: "Item was successfully upd
    else
      render :edit, status: :unprocessable_entity
    end
  end

  private

  # All the previous code

  def set_line_item
    @line_item = @line_item_date.line_items.find(params[:id])
  end
end
```

We know that we will need the `set_line_item` callback for the `#destroy` action as well, so we can anticipate and add it to the list of actions requiring this callback.

Now that our `#edit` and `#update` actions are implemented, let's add the `LineItems#edit` view to be able to test in the browser:

```
<%# app/views/line_items/edit.html.erb %>

<main class="container">
  <%= link_to sanitize("&larr; Back to quote"), quote_path(@quote) %>

  <div class="header">
    <h1>Edit item</h1>
  </div>

  <%= render "form",
             quote: @quote,
             line_item_date: @line_item_date,
             line_item: @line_item %>
</main>
```

As we can notice, the `LineItems#edit` view is very similar to the `LineItems#new` view. Only the title changes. As we already built the form in

the previous section, we are ready to experiment in the browser. Everything works as expected; only one more action to go!

Deleting line items without Turbo

The `#destroy` action is the simplest of all five as it doesn't require a view. We only need to delete the **line item** and then redirect to the `Quotes#show` page:

```
# app/controllers/line_items_controller.rb

class LineItemsController < ApplicationController
  # All the previous code

  def destroy
    @line_item.destroy

    redirect_to quote_path(@quote), notice: "Item was successfully destr
  end

  # All the previous code
end
```

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

Our CRUD controller is now working, but we now want all interactions to happen on the same page. Thanks to the power of Turbo, it will only require a few lines of code to slice our page into pieces that can be updated independently.

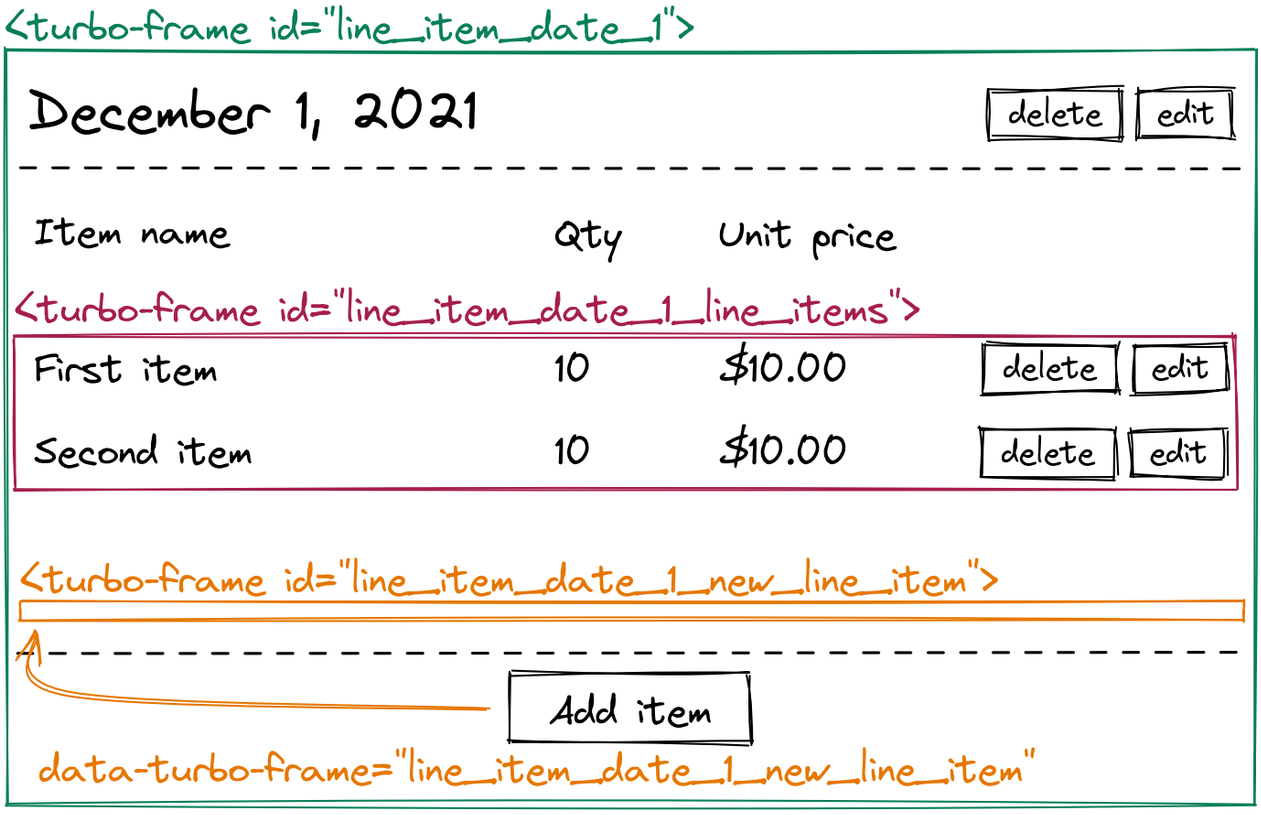
Adding Turbo Frames and Turbo Streams

Creating line items with Turbo

Now that our CRUD controller is working as expected, it's time to improve the user experience so that all the interactions happen on `Quotes#show` page.

To be clear on the requirements, let's first sketch the desired behavior. ***From now on, we will zoom in sketches on a single line item date for them to remain readable.***

When a user visits the `Quotes#show` page and clicks on the "Add item" button **for a specific date**, we want the form to appear on the `Quotes#show` page right above the "Add item" button we just clicked on. We will do this with Turbo Frames, of course! To make it work, we have to connect the "Add item" link to an empty Turbo Frame thanks to the `data-turbo-frame` data attribute.



Sketch of a specific date with Turbo Frames

For Turbo to properly replace the Turbo Frame on the `Quotes#show` page, the Turbo Frame on the `LineItems#new` page must have the same id.

We notice that the Turbo Frame ids are longer than in the previous chapters. **Turbo Frames must have unique ids on the page to work properly**. As we have multiple dates on the page, if the id for the empty Turbo Frame was only `new_line_item`, or the id for the list of **line items** was only `line_items`, we would have multiple Turbo Frames with the same id.

Let's explain why **Turbo Frames on the same page must have different ids**. If we did like in previous chapters, our `create.turbo_stream.erb` view would look like this:

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

<%= turbo_stream.update LineItem.new, "" %>
<%= turbo_stream.append "line_items", @line_item %>
```

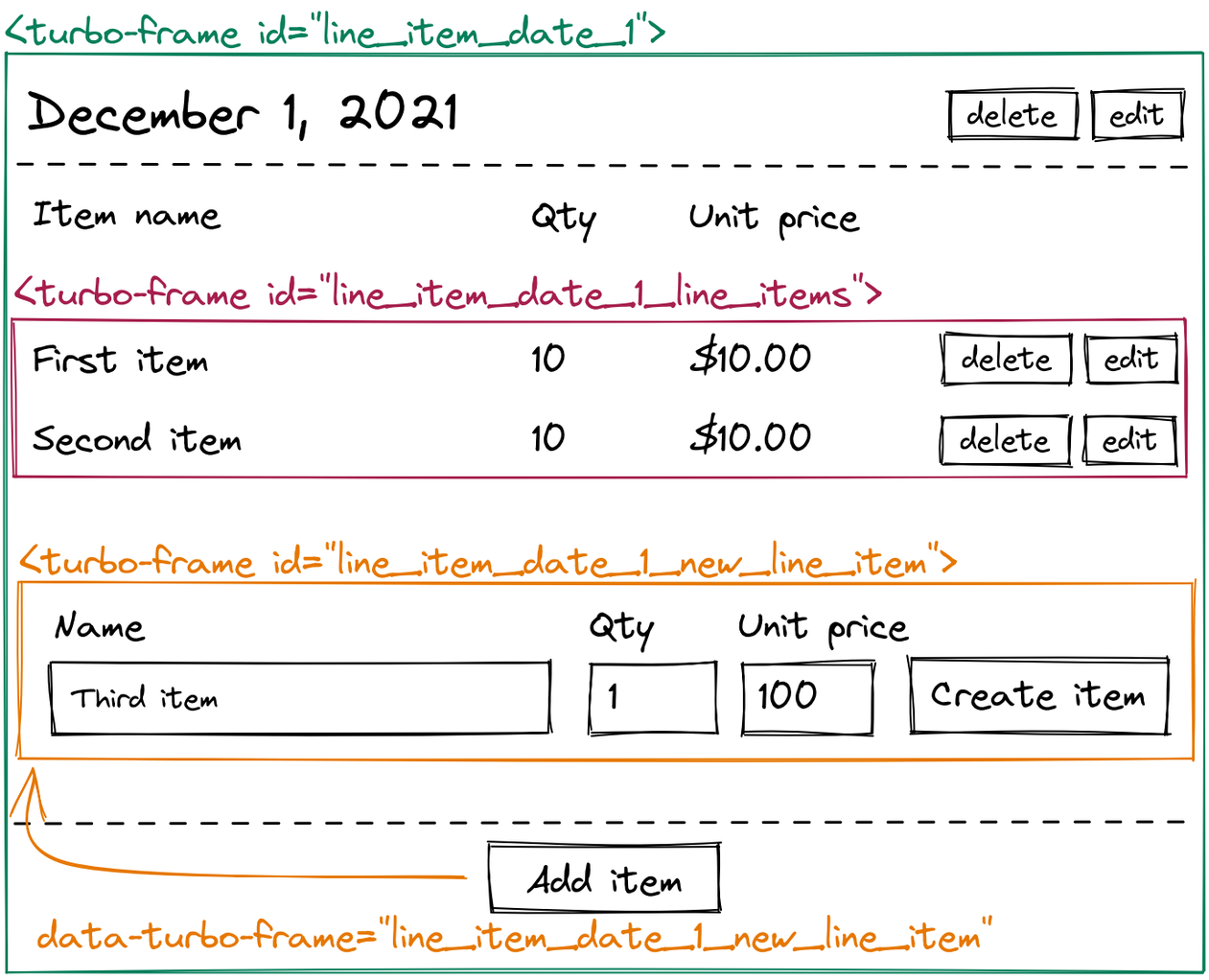
If there are multiple **line item dates** on the quote, then there would be the `new_line_item` and the `line_items` ids multiple times on the `Quotes#show` page. How could Turbo guess what to do if there are multiple times the same id? Our created **line item** would probably be *appended* in the list of line items of the wrong date!

A good convention is to prefix the ids we would normally have by the `dom_id` of the parent resource to solve this issue. That way, we are sure our ids are unique.

For Turbo to work properly, we need a Turbo Frame of the same id on the `LineItems#new` page:

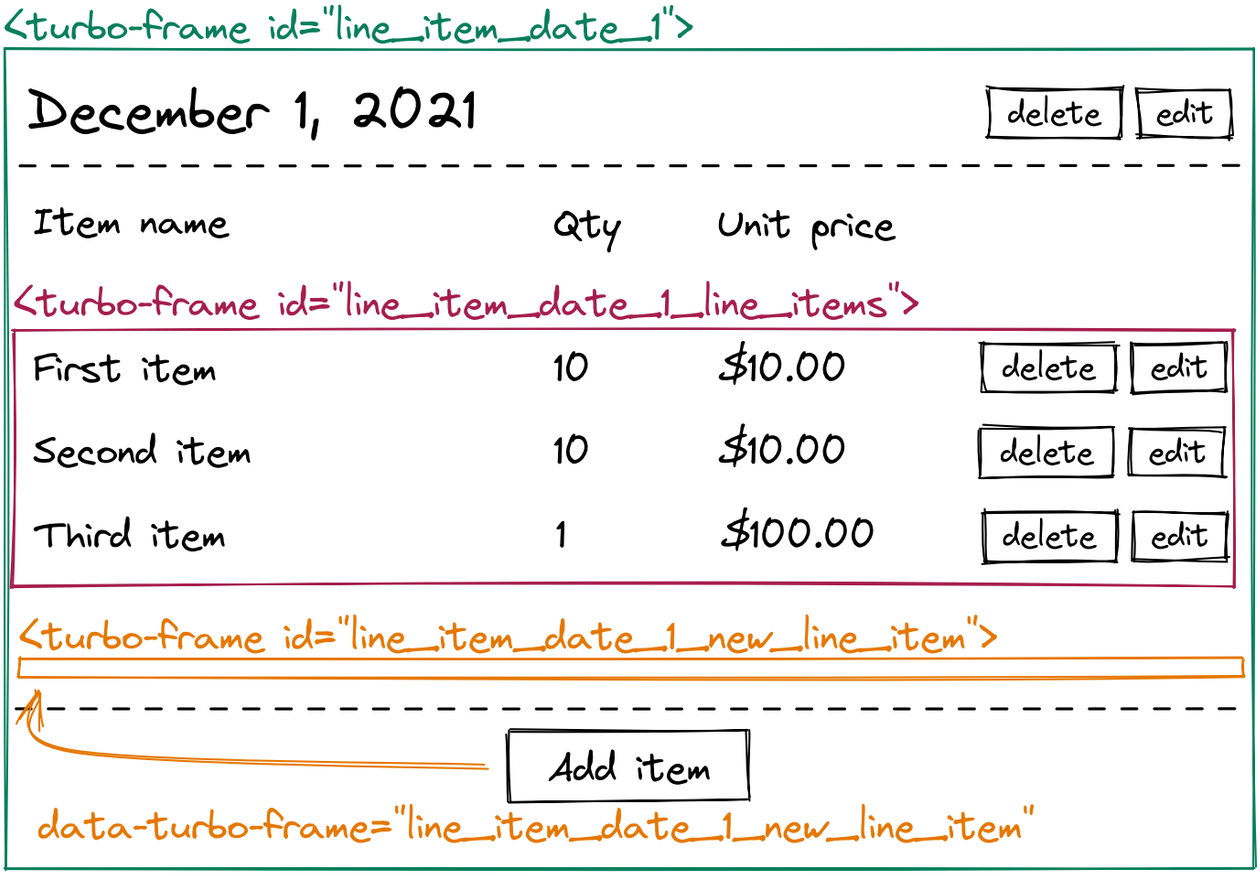
Sketch of the LineItems#new page with a Turbo Frame

With those Turbo Frames in place, when a user clicks on the "New item" button, Turbo will successfully replace the empty Turbo Frame on the `Quotes#show` page with the Turbo Frame containing the form on the `LineItems#new` page:



Sketch of a specific date with the form from the LineItems#new page

When the user submits the form, we want the created **line item** to be *appended* to the list of **line items** for this specific date:



Sketch of a specific date with the created item appended

Now that the requirements are clear, it should only take a few lines of code to make it real, thanks to the power of Turbo Frames and Turbo Streams!

Let's start working on the first part: making the form appear on the `Quotes#show` page when a user clicks on the "Add item" button. On each **line item date**, let's add an empty Turbo Frame and connect the "Add date" button to it:

```
<%# app/views/line_item_dates/_line_item_date.html.erb %>

<%= turbo_frame_tag line_item_date do %>
  <div class="line-item-date">
    <!-- All the previous code -->
    <div class="line-item-date__body">
      <div class="line-item line-item--header">
        <!-- All the previous code -->
      </div>

      <%= render line_item_date.line_items, quote: quote, line_item_date

      <%= turbo_frame_tag dom_id(LineItem.new, dom_id(line_item_date)) %

      <div class="line-item-date__footer">
        <%= link_to "Add item",
                    [:new, quote, line_item_date, :line_item],
                    data: { turbo_frame: dom_id(LineItem.new, dom_id(lin
                    class: "btn btn--primary" %>
      </div>
    </div>
  </div>
<% end %>
```

As mentioned above, for nested resources, we want to prefix the `dom_id` of the resource with the `dom_id` of the parent. The `dom_id` helper takes an optional prefix as a second argument. We could use the `dom_id` helper to follow our convention:

```
line_item_date = LineItemDate.find(1)

dom_id(LineItem.new, dom_id(line_item_date))
# => line_item_date_1_new_line_item
```

This approach works fine, but it is hard to read. It also has an edge case:

```
dom_id("line_items", dom_id(line_item_date))
# This does not return "line_item_date_1_line_items"
# It raises an error as "line_items" does not respond to `#to_key`
# and so can't be transformed into a dom_id
```

Instead of relying on the `dom_id` helper directly, let's create a helper to make our ids easier to generate/read and ensure all developers in our team will use the same convention:

```
# app/helpers/application_helper.rb

module ApplicationHelper
  # All the previous code

  def nested_dom_id(*args)
    args.map { |arg| arg.respond_to?(:to_key) ? dom_id(arg) : arg }.join
  end
end
```

With this helper in place, it is much easier to generate and read our `dom_ids`:

```
line_item_date = LineItemDate.find(1)

nested_dom_id(line_item_date, LineItem.new)
# => line_item_date_1_new_line_item

nested_dom_id(line_item_date, "line_items")
# => line_item_date_1_line_items
```

Let's just update the view to use our new convention:

```
<%# app/views/line_item_dates/_line_item_date.html.erb %>

<%= turbo_frame_tag line_item_date do %>
  <div class="line-item-date">
    <!-- All the previous code -->
    <div class="line-item-date__body">
      <div class="line-item line-item--header">
        <!-- All the previous code -->
      </div>
```

```
      <%= render line_item_date.line_items, quote: quote, line_item_date

      <%= turbo_frame_tag nested_dom_id(line_item_date, LineItem.new) %>

      <div class="line-item-date__footer">
        <%= link_to "Add item",
                    [:new, quote, line_item_date, :line_item],
                    data: { turbo_frame: nested_dom_id(line_item_date, L
                    class: "btn btn--primary" %>
      </div>
    </div>
  </div>
<% end %>
```

Now that our Turbo Frames have the expected ids on the `Quotes#show` page, we need to have matching Turbo Frames on the `LineItems#new` page for Turbo to swap the frames. Let's wrap our form in a Turbo Frame tag:

```
<%# app/views/line_items/new.html.erb %>

<main class="container">
  <%= link_to sanitize("&larr; Back to quote"), quote_path(@quote) %>

  <div class="header">
    <h1>New item for <%= l(@line_item_date.date, format: :long) %></h1>
  </div>

  <%= turbo_frame_tag nested_dom_id(@line_item_date, LineItem.new) do %>
    <%= render "form",
               quote: @quote,
               line_item_date: @line_item_date,
               line_item: @line_item %>
  <% end %>
</main>
```

Let's experiment in the browser. When clicking on the "Add item" button the form should appear at the expected position for this date!

As in the previous chapters, when we submit an **invalid** form, the errors appear on the page as expected.

We have to give Turbo more precise instructions when submitting a **valid** form thanks to a Turbo Stream view. We want to perform two actions:

1. Remove the form we just submitted from the DOM
2. Append the created **line item** to the list of **line items** for this specific date

Let's edit our `LineItemsController#create` action to respond to the `turbo_stream` format:

```
# app/controllers/line_items_controller.rb

class LineItemsController < ApplicationController
  # All the previous code...

  def create
    @line_item = @line_item_date.line_items.build(line_item_params)

    if @line_item.save
      respond_to do |format|
        format.html { redirect_to quote_path(@quote), notice: "Item was
        format.turbo_stream { flash.now[:notice] = "Item was successfull
      end
    else
      render :new, status: :unprocessable_entity
    end
  end

  # All the previous code...
end
```

Let's create our view that will perform the two actions that we want:

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

<%# Step 1: empty the Turbo Frame containing the form %>
<%= turbo_stream.update nested_dom_id(@line_item_date, LineItem.new), ""

<%# Step 2: append the created line item to the list %>
<%= turbo_stream.append nested_dom_id(@line_item_date, "line_items") do
  <%= render @line_item, quote: @quote, line_item_date: @line_item_date
<% end %>
```

```
<%= render_turbo_stream_flash_messages %>
```

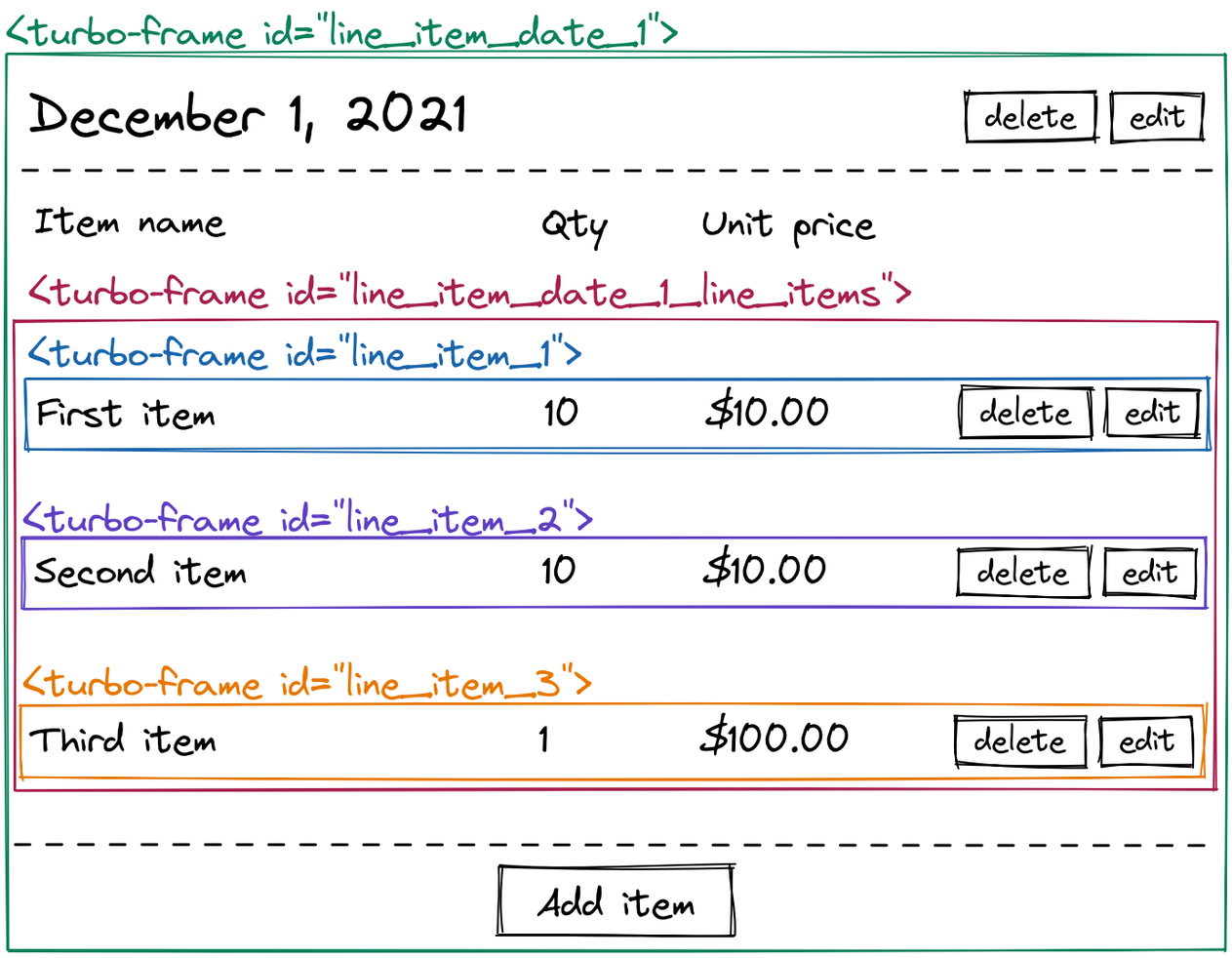
The last thing we need to do is to add a Turbo Frame to wrap the list of **line items** for each specific date:

```
<%# app/views/line_item_dates/_line_item_date.html.erb %>

<!-- All the previous code -->
<%= turbo_frame_tag nested_dom_id(line_item_date, "line_items") do %>
  <%= render line_item_date.line_items, quote: quote, line_item_date: li
<% end %>
<!-- All the previous code -->
```

Let's test it in our browser, and everything should work as expected. That was a lot of work! We are almost there. The `#edit`, `#update`, and `#destroy` actions will be easier to implement now that nearly everything is in place.

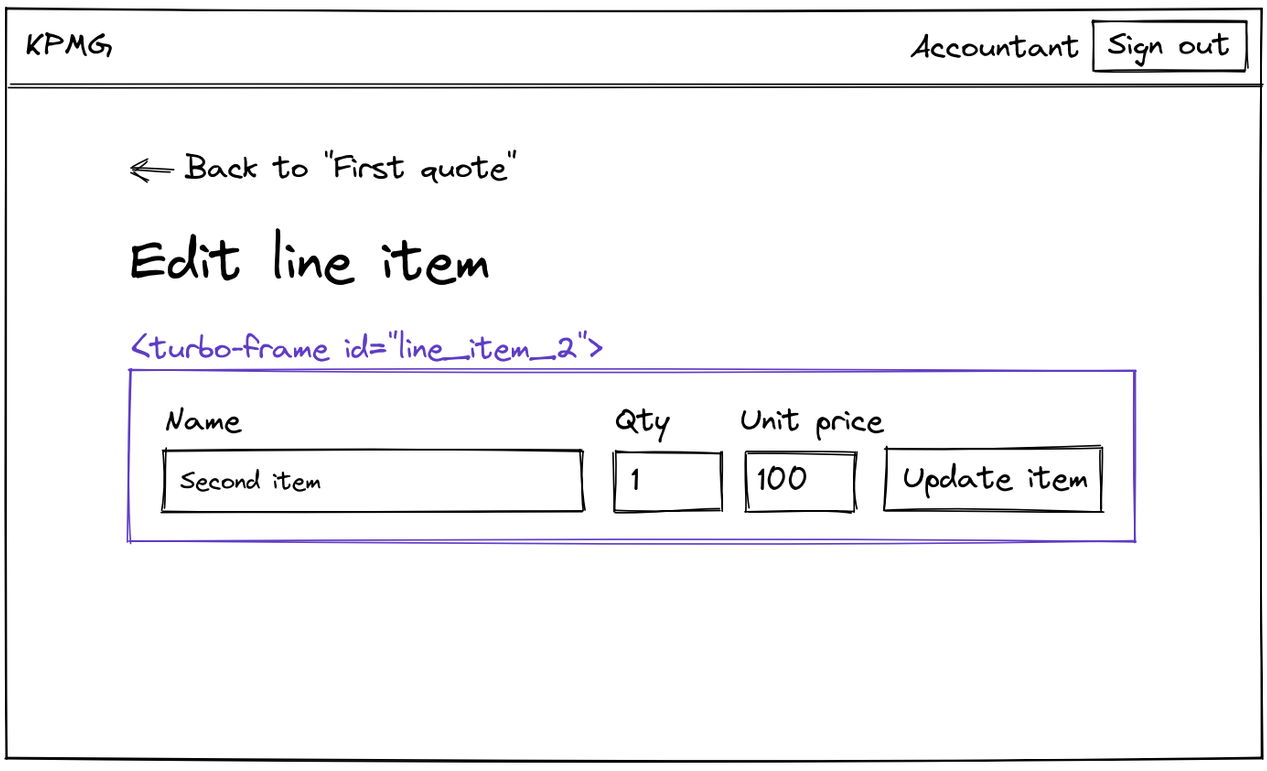
Updating line items with Turbo

Like we did for the `#new` and `#create` actions, we want to make the `#edit` and `#update` actions for a quote happen on the `Quotes#show` page. We already have most of the Turbo Frames we need, but we are going to need each line item also to be wrapped inside a Turbo Frame as described in the sketch below:



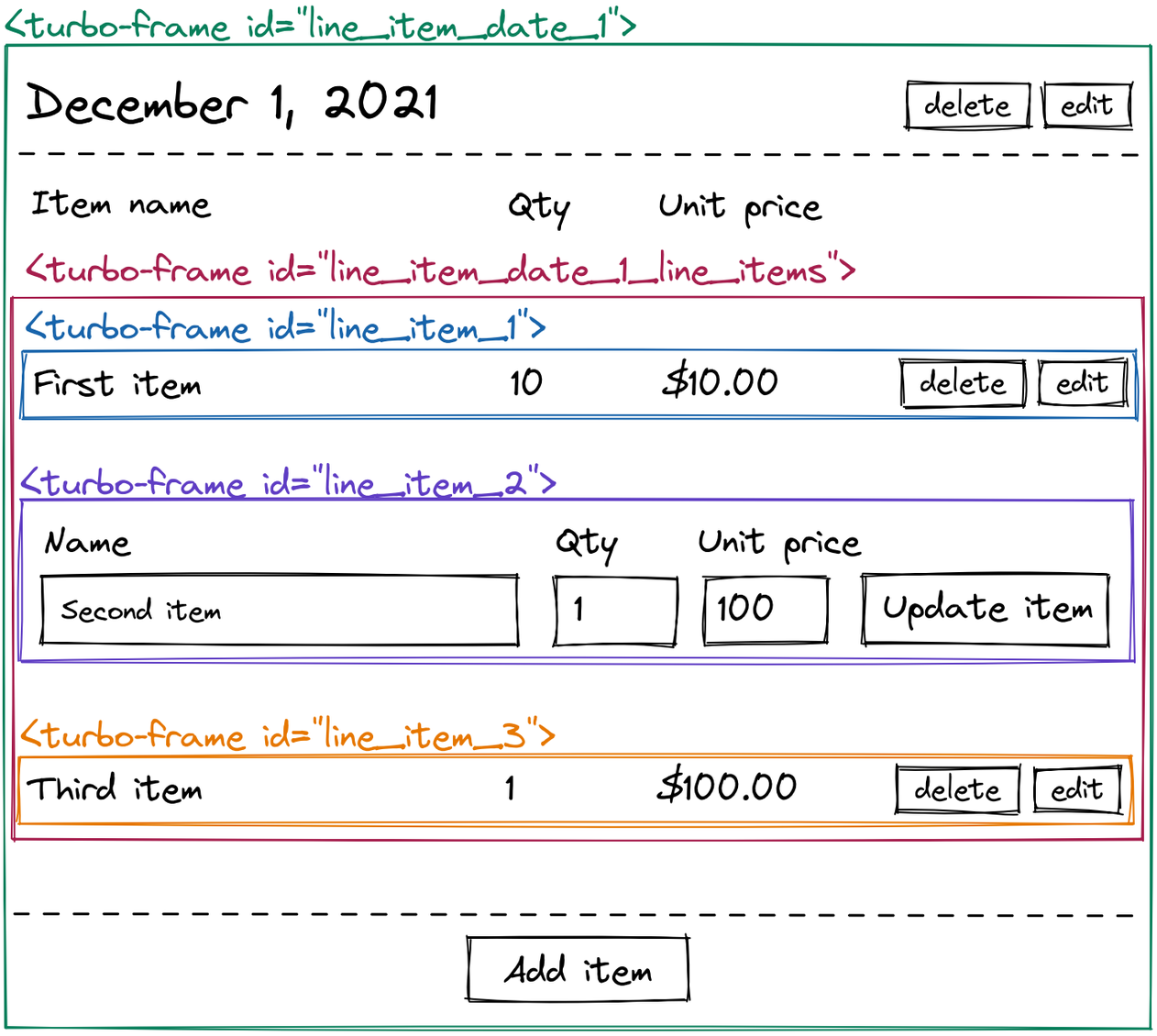
Sketch of a specific date with Turbo Frames

When clicking on the "Edit" link for the **second line item** of our sketch that is within a Turbo Frame of id `line_item_2`, Turbo expects to find a Turbo Frame with the same id on the `LineItems#edit` page as described in the sketch below:



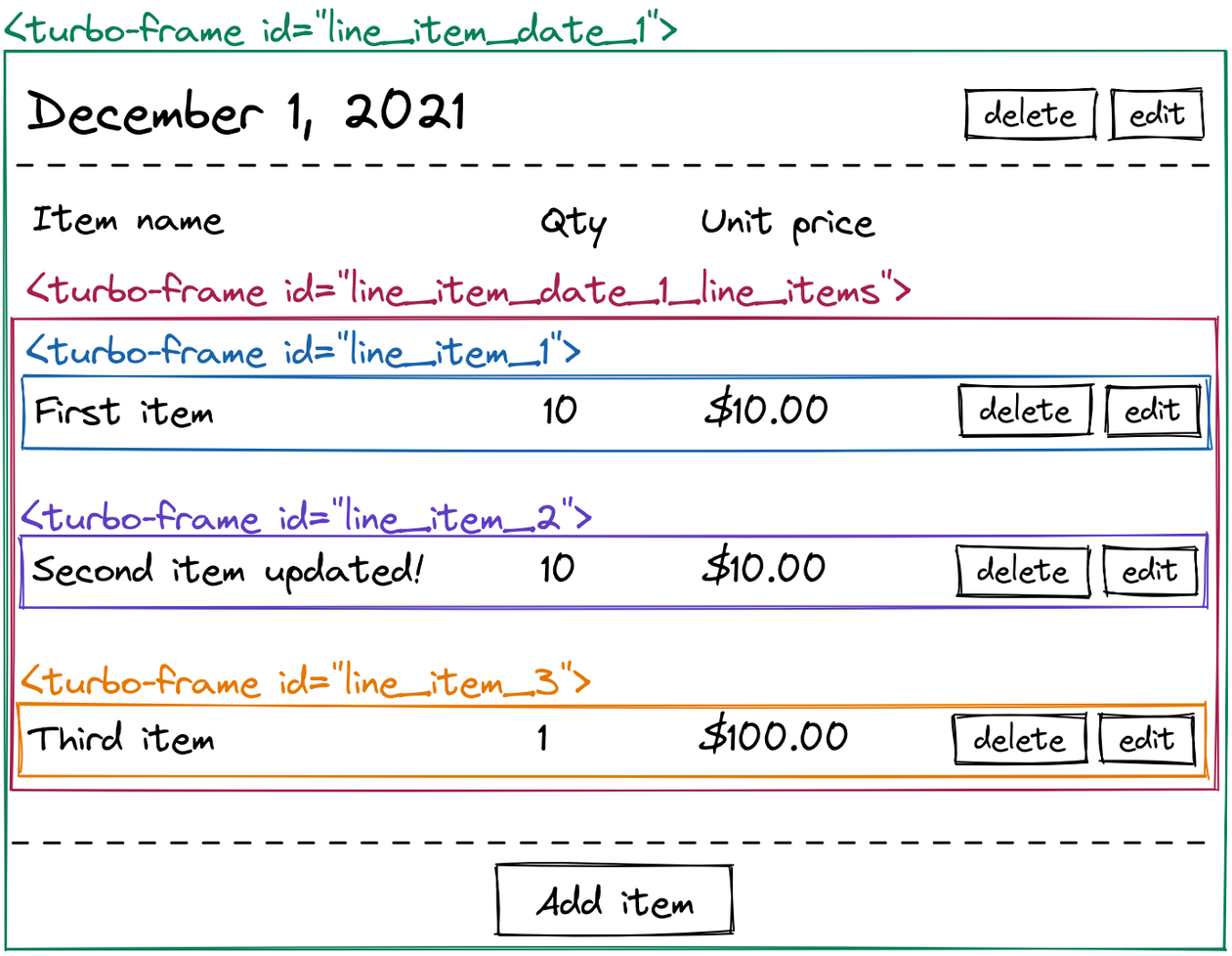
Sketch of the LineItems#edit page with a Turbo Frame of the same id

With those Turbo Frames in place, Turbo will be able to replace the line item with the form from the `LineItems#edit` page when clicking on the "Edit" link of a line item:



Sketch of specific date with the edit form for the second item

When submitting the form, we want the form to be replaced with the final quote:



Sketch of the specific date after the form was submitted

Now that the requirements are clear, it's time to start coding. The first part of the job is to make the edit form successfully replace the HTML of **line items** on the `Quotes#show` page. To do this, let's wrap every item in a Turbo Frame:

```
<%# app/views/line_items/_line_item.html.erb %>

<%= turbo_frame_tag line_item do %>
  <div class="line-item">
    <!-- All the previous code -->
  </div>
<% end %>
```

We also need to remove the `data-turbo-frame="_top"` data attribute from the "Edit" link:

```
<%# app/views/line_items/_line_item.html.erb %>

<!-- All the previous code -->
<%= link_to "Edit",
            [:edit, quote, line_item_date, line_item],
            class: "btn btn--light" %>
<!-- All the previous code -->
```

Now that we wrapped our **line items** in Turbo Frames, we need to wrap the form in the `LineItems#edit` page in a Turbo Frame as well:

```
<%# app/views/line_items/edit.html.erb %>

<main class="container">
  <%= link_to sanitize("&larr; Back to quote"), quote_path(@quote) %>

  <div class="header">
    <h1>Edit item</h1>
  </div>

  <%= turbo_frame_tag @line_item do %>
    <%= render "form",
               quote: @quote,
               line_item_date: @line_item_date,
               line_item: @line_item %>
  <% end %>
</main>
```

With this Turbo Frame in place, we can test in the browser. When clicking on the "Edit" button on a **line item**, the form successfully replaces the **line item** on the `Quotes#show` page.

If we submit an invalid form, everything already works as expected. If we submit a valid form, the line item date is successfully replaced but we miss the flash message. To solve this, we will need a Turbo Stream view. Let's first enable our controller to render a Turbo Stream view:

```
# app/controllers/line_items_controller.rb

def update
  if @line_item.update(line_item_params)
    respond_to do |format|
      format.html { redirect_to quote_path(@quote), notice: "Item was su
      format.turbo_stream { flash.now[:notice] = "Item was successfully
    end
  else
    render :edit, status: :unprocessable_entity
  end
end
```

Let's now create the `update.turbo_stream.erb` view to replace the line item form with the line item partial and render the flash message:

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

<%= turbo_stream.replace @line_item do %>
  <%= render @line_item, quote: @quote, line_item_date: @line_item_date
<% end %>

<%= render_turbo_stream_flash_messages %>
```

Let's test it in the browser; everything should work as expected!

Destroying line items with Turbo

The last feature we need is the ability to remove line item dates from our quote. To do this, we first have to support the Turbo Stream format in the `#destroy` action in the controller:

```
# app/controllers/line_items_controller.rb

def destroy
  @line_item.destroy

  respond_to do |format|
    format.html { redirect_to quote_path(@quote), notice: "Date was succ
    format.turbo_stream { flash.now[:notice] = "Date was successfully de
  end
end
```

In the view, we only have to remove the line item and render the flash message:

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

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

Let's not forget to remove the `data-turbo-frame="_top"` data attribute from the "Delete" button:

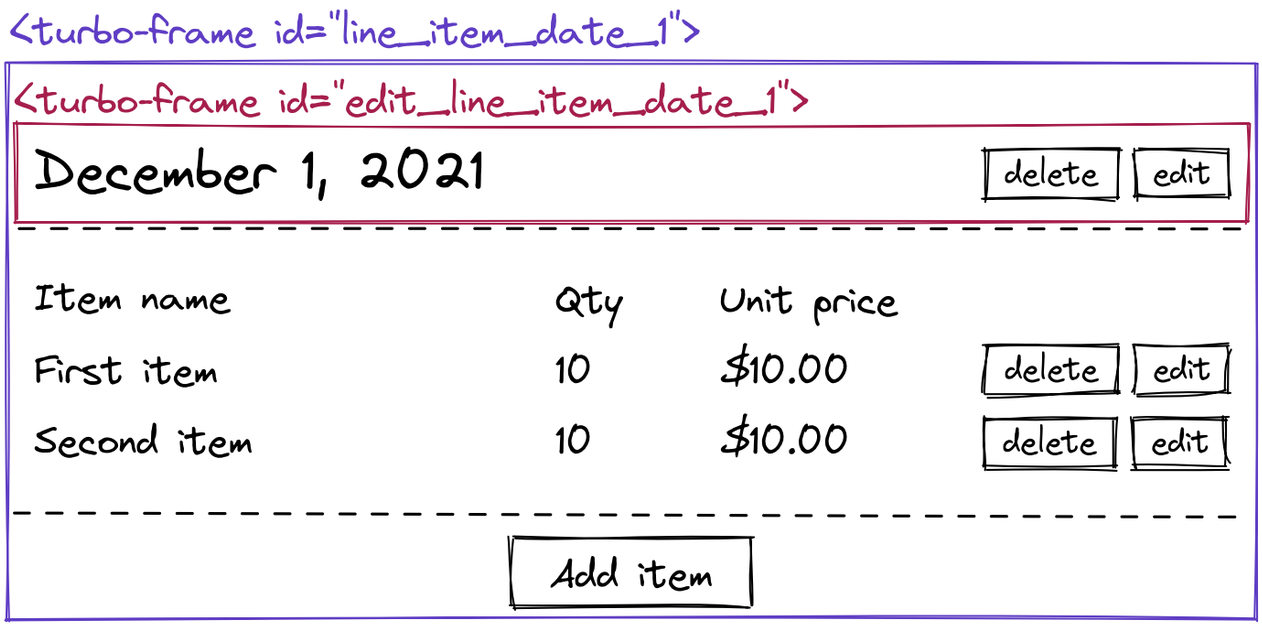
```erb
<%# app/views/line_items/_line_item.html.erb %>

<!-- All the previous code -->
<%= button_to "Delete",
              [quote, line_item_date, line_item],
              method: :delete,
              class: "btn btn--light" %>
<!-- All the previous code -->
```

We can finally test in our browser that everything works as expected. The behavior is almost exactly the same as the one we had for quotes!

Editing line item dates with Turbo

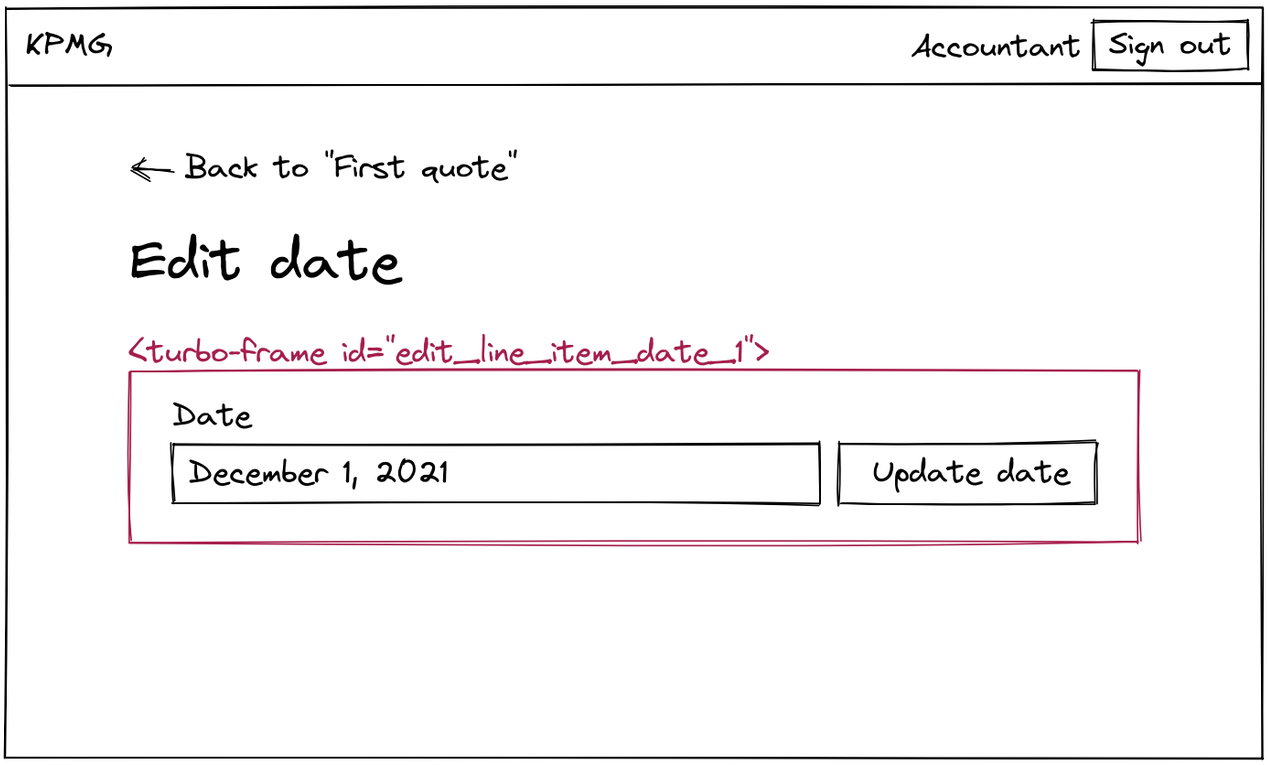
The actions for our line items are now complete! However, we have a small issue: the whole **line item date** card is replaced by the edit form when clicking on the "Edit" link for a **line item date**. We would like only the card's header contaning the date to be replaced.

Let's wrap only the header of the **line item date** card inside another Turbo Frame with a unique id by prefixing its `dom_id` with "edit":



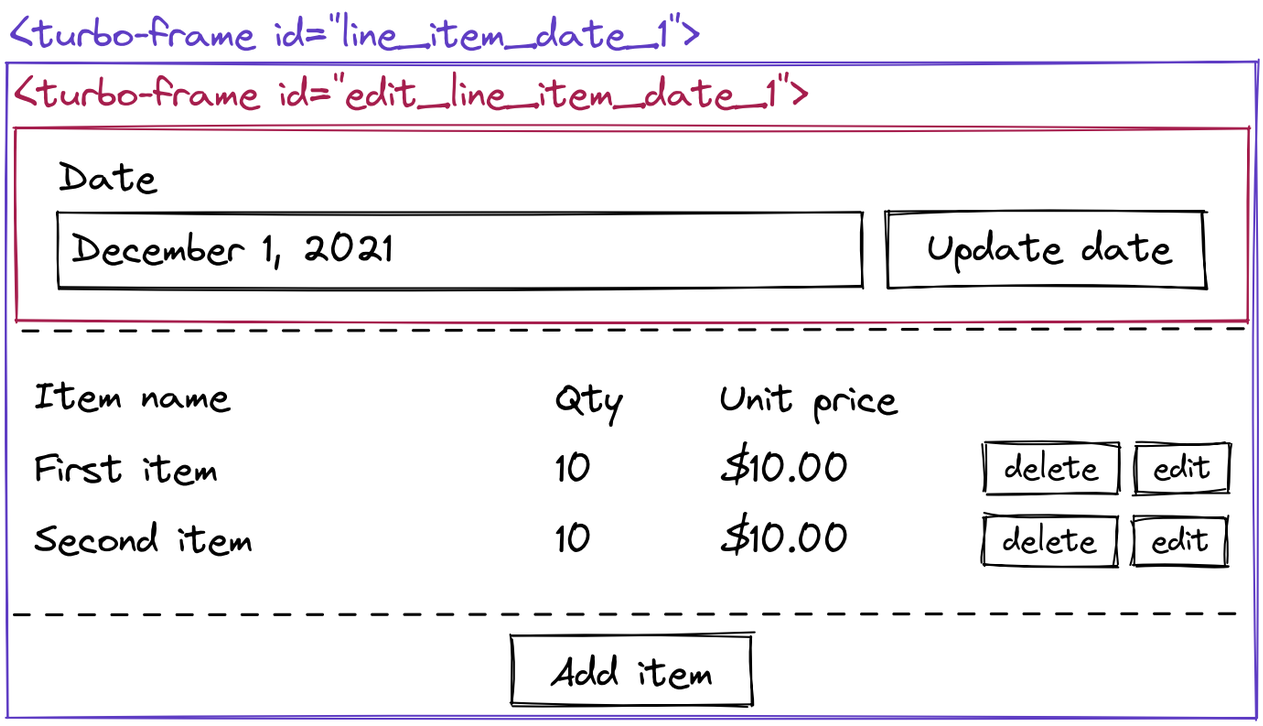
Sketch of a specific date on the Quotes#show page

For Turbo to be able to replace the Turbo Frame, we need a Turbo Frame with the same id on the `LineItemDates#edit` page:



Sketch of the LineItemDates#edit page with a Turbo Frame

Now, when clicking on the "Edit" button for this specific date, Turbo will only replace the header of the **line item date** card:



Sketch of the LineItemDates#edit page with a Turbo Frame

Now that the requirements are clear, it's time to start coding. Let's first start by adding the Turbo Frame with the "edit" prefix to the line item date partial:

```
<%# app/views/line_item_dates/_line_item_date.html.erb %>

<%= turbo_frame_tag line_item_date do %>
  <div class="line-item-date">
    <%= turbo_frame_tag dom_id(line_item_date, :edit) do %>
```

```
      <div class="line-item-date__header">
        <!-- All the previous code -->
      </div>
    <% end %>
    <div class="line-item-date__body">
      <!-- All the previous code -->
    </div>
  </div>
<% end %>
```

We also need to add the "edit" prefix to the `LineItemDates#edit` page:

```
<%# app/views/line_item_dates/edit.html.erb %>

<main class="container">
  <%= link_to sanitize("&larr; Back to quote"), quote_path(@quote) %>

  <div class="header">
    <h1>Edit date</h1>
  </div>

  <%= turbo_frame_tag dom_id(@line_item_date, :edit) do %>
    <%= render "form", quote: @quote, line_item_date: @line_item_date %>
  <% end %>
</main>
```

Let's test it in the browser. Now when clicking on the edit link for a **line item date**, only the card's header is replaced by the form from the `LineItemDates#edit` page.

Preserving state with Turbo Rails

Until now, we managed to preserve the *state* of our application all the time by making pieces of our page really independent. However, there is a small glitch in our application right.

To demonstrate our *state* issue, let's navigate on the `Quotes#show` page for the first quote and open a few **line item** forms for the first **line item date** by clicking on the "Edit" button for those **line items**. Let's then update the first **line item date**. The forms are now closed again!

This is because to keep our dates in ascending order, we completely remove the **line item date** card from the DOM and then re-attach it at the correct position in the list. Of course, if we completely remove and re-attach the **line item date**, we will lose the *state* of the **line items** within this date as the partial is rendered with all forms closed by default.

Here we reached the limits of what we can do with Turbo Rails without writing any custom JavaScript. If we wanted to preserve the *state* of the application on the `Quotes#show` page when updating a **line item date**, we would have two solutions:

- Don't reorder the items when using the Turbo Stream format
- Reorder items in the front-end with a Stimulus controller

Even if this is a minor glitch, knowing the limitations of what Turbo can do on its own is important! In this tutorial, we will simply ignore this glitch.

Testing our code with system tests

Our work wouldn't be complete if we didn't add a few tests. We should always write at least system tests to ensure the happy path is covered. If we make a mistake, we can correct it before pushing our code into production.

Let's add a system test file to test the happy path of the CRUD on our line items:

```
# test/system/line_items_test.rb

require "application_system_test_case"

class LineItemSystemTest < ApplicationSystemTestCase
  include ActionView::Helpers::NumberHelper

  setup do
    login_as users(:accountant)

    @quote          = quotes(:first)
    @line_item_date = line_item_dates(:today)
    @line_item      = line_items(:room_today)

    visit quote_path(@quote)
```

```
end

test "Creating a new line item" do
  assert_selector "h1", text: "First quote"

  within "##{dom_id(@line_item_date)}" do
    click_on "Add item", match: :first
  end
  assert_selector "h1", text: "First quote"

  fill_in "Name", with: "Animation"
  fill_in "Quantity", with: 1
  fill_in "Unit price", with: 1234
  click_on "Create item"

  assert_selector "h1", text: "First quote"
  assert_text "Animation"
  assert_text number_to_currency(1234)
end

test "Updating a line item" do
  assert_selector "h1", text: "First quote"

  within "##{dom_id(@line_item)}" do
    click_on "Edit"
  end
  assert_selector "h1", text: "First quote"

  fill_in "Name", with: "Capybara article"
  fill_in "Unit price", with: 1234
  click_on "Update item"

  assert_text "Capybara article"
  assert_text number_to_currency(1234)
end

test "Destroying a line item" do
  within "##{dom_id(@line_item_date)}" do
    assert_text @line_item.name
  end

  within "##{dom_id(@line_item)}" do
    click_on "Delete"
  end
```

```
    within "##{dom_id(@line_item_date)}" do
      assert_no_text @line_item.name
    end
  end
end
```

If we run the `bin/rails test:all` command, we will notice that we have two previous tests to fix. As we have too many "Edit" and "Delete" links with the same name, Capybara won't know which one to click on and raise a `Capybara::Ambiguous` error.

To fix that issue, we have to be more specific with the ids we use in our `within` blocks:

```
# test/system/line_item_dates_test.rb

# All the previous code

test "Updating a line item date" do
  assert_selector "h1", text: "First quote"

  within id: dom_id(@line_item_date, :edit) do
    click_on "Edit"
  end

  assert_selector "h1", text: "First quote"

  fill_in "Date", with: Date.current + 1.day
  click_on "Update date"

  assert_text I18n.l(Date.current + 1.day, format: :long)
end

test "Destroying a line item date" do
  assert_text I18n.l(Date.current, format: :long)

  accept_confirm do
    within id: dom_id(@line_item_date, :edit) do
      click_on "Delete"
    end
  end

  assert_no_text I18n.l(Date.current, format: :long)
```

```
end

# All the previous code
```

Let's run all the tests with `bin/rails test:all` command, and they should now all be green!

Wrap up

In this chapter, we almost finalized our quote editor. We learned how to manage nested Turbo Frames and keep our code readable thanks to naming conventions on Turbo Frames!

In the next chapter, we will completely finalize our quote editor! See you there!

← previous next →