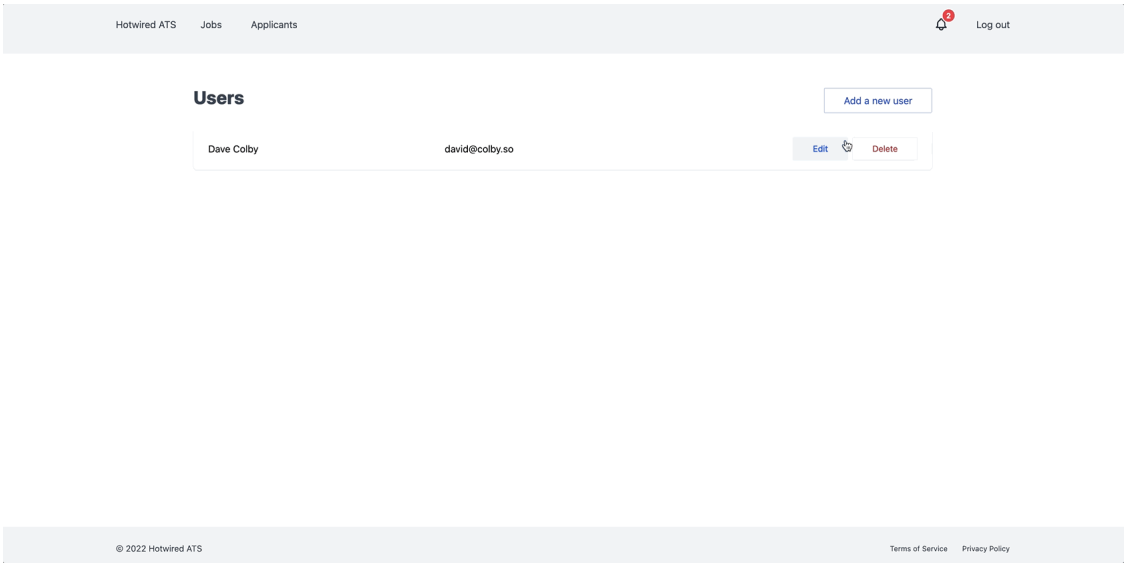
User management

Our application, like many B2B SaaS applications, is most valuable when used by a team of people with access to the same set of data. Right now, every user in our application is siloed within their own account, with no way to add users to collaborate with.

In this chapter, we will add a user management interface enabling users to create, edit, and delete additional users within their account. New users created in an existing account will receive an email inviting them to set a password and sign in.

This chapter will use the tools that we have used throughout this book, with one additional Turbo Frame technique used to create a simple inline editing experience.

When we are finished, our user management page will work like this:



User management

We will start by setting up a new `UsersController` to handle user management in our application.

As usual, generate that controller from your terminal:

```
rails g controller Users
```

Update `config/routes.rb`:

config/routes.rb

```
9 | resources :users
```

Update the `UsersController`:

app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  before_action :authenticate_user!
  before_action :set_user, only: %i[edit update
destroy]

  def index
    @users = User.where(account_id:
current_user.account_id)
  end

  def new; end

  def create; end

  def edit; end

  def update; end

  def destroy; end
```

```
21
22
23
24
```

```
  private

  def set_user
    @user = User.find(params[:id])
  end
end
```

Here, we implemented the `index` action and stubbed out the rest of the actions we will need. We will define the rest of the actions as we work through this chapter.

Let's build out the index page and add the ability to create new users using our standard slideover drawer. From your terminal, add the views:

```
touch app/views/users/index.html.erb
touch app/views/users/_form.html.erb
touch app/views/users/_user.html.erb
```

Fill in the index view:

app/views/users/index.html.erb

```
<div class="flex items-baseline justify-between mb-6">
  <h2 class="mt-6 text-3xl font-extrabold text-gray-
700">
    Users
  </h2>
  <%= link_to "Add a new user", new_user_path, class:
"btn-primary-outline", data: { action: "click-
>slideover#open", remote: true } %>
</div>
<div class="shadow overflow-hidden sm:rounded-md">
  <div class="divide-y divide-gray-200 flex flex-col"
id="users">
    <%= render @users %>
```

```
  </div>
</div>
```

Here, we are rendering a collection of users (defaulting to the `user` partial for each user in the collection) and connecting the `slideover#open` action to the Add new user button.

Fill in the user partial:

app/views/users/_user.html.erb

```
<%= turbo_frame_tag dom_id(user) do %>
  <div class="flex flex-col md:flex-row items-center
px-4 py-4 sm:px-6 justify-between">
    <div class="w-1/3">
      <%= user.name %>
      <div class="flex space-x-2 text-gray-500 text-
sm">
        <div>
          <!-- Extra info will go here -->
        </div>
      </div>
    </div>
    <div class="w-1/3">
      <%= user.email %>
    </div>
    <div class="flex justify-end w-1/3">
      <%= link_to "Edit", edit_user_path(user), class:
"btn border border-blue-200 hover:bg-blue-200 text-sm
text-blue-700 mr-2" %>
      <%= link_to "Delete", user_path(user), method:
:delete, class: "btn border border-red-200 hover:bg-
red-100 text-sm text-red-700", data: { confirm: "Are
you sure?" } %>
    </div>
  </div>
<% end %>
```

Mostly regular old Rails here. Take note that each user will be wrapped in their own unique Turbo Frame. This means that when the Edit button is clicked, the navigation will be scoped within the Turbo Frame for the user.

This scoped navigation is how we will enable inline editing later in this chapter. Before we take on inline editing, we are referencing an undefined `name` method in the view. Define that in `app/models/user.rb`:

app/models/user.rb

```
17  def name
18    [first_name, last_name].join(' ').presence || '(Not
19  set)'
    end
```

When a user signs up for a new account on the Devise registration page we do not ask them for their name, so the name method returns either the user's first and last name or a placeholder string.

Fill in the form partial, which we will open in a slideover:

app/views/users/_form.html.erb

```
<%= form_with(model: user, id: 'user-form', html: {
class: "space-y-6" }, data: { remote: true }) do |form|
%>
  <% if user.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize(user.errors.count, "error") %>
prohibited this user from being saved:</h2>

      <ul>
        <% user.errors.each do |error| %>
          <li><%= error.full_message %></li>
        <% end %>
      </ul>
    </div>
  <% end %>
```

```
  <div class="form-group">
    <%= form.label :first_name %>
    <div class="mt-1">
      <%= form.text_field :first_name %>
    </div>
  </div>

  <div class="form-group">
    <%= form.label :last_name %>
    <div class="mt-1">
      <%= form.text_field :last_name %>
    </div>
  </div>

  <div class="form-group">
    <%= form.label :email %>
    <div class="mt-1">
      <%= form.text_field :email %>
    </div>
  </div>

  <%= form.submit 'Submit', class: 'btn-primary float-right' %>
<% end %>
```

This is our standard slideover-powered form partial, nothing exciting to note.

Finish up new user creation by updating the `UsersController`:

app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  before_action :authenticate_user!
  before_action :set_user, only: %i[edit update destroy]

  def index
    @users = current_user.account.users
  end
```

```
  def new
    @user = User.new
    html = render_to_string(partial: 'form', locals: {
user: @user })
    render operations: cable_car
      .inner_html('#slideover-content', html: html)
      .text_content('#slideover-header', text: 'Add a
new user')
  end

  def create
    @user = User.new(user_params)
    @user.account = current_user.account
    @user.password = SecureRandom.alphanumeric(24)

    if @user.save
      html = render_to_string(partial: 'user', locals:
{ user: @user })
      render operations: cable_car
        .prepend('#users', html: html)
        .dispatch_event(name: 'submit:success')
    else
      html = render_to_string(partial: 'form', locals:
{ user: @user })
      render operations: cable_car
        .inner_html('#user-form', html: html), status:
:unprocessable_entity
    end
  end

  def edit; end

  def update; end

  def destroy; end

  private

  def set_user
```

```
    @user = User.find(params[:id])
  end

  def user_params
    params.require(:user).permit(:first_name,
:last_name, :email)
  end
end
```

Here, we updated the `new` and `create` actions using our now standard CableReady operations.

One thing to note is the generation of a random password for the user. Devise requires a password to persist a user in the database, so we set a random password for the user in the `create` action.

The new user will never use this random password. Later in this chapter, we will build a form new users will use to activate their account and set their own password.

Before we get to that, let's learn something new by building the inline editing experience on the user index page.

Editing users with turbo frames

Since our users only have three editable fields (first name, last name, and email address) right now, opening up a drawer to edit user information is not strictly necessary. As an alternative, we can render an edit form directly in the users list using our new Turbo skills.

To begin, we will need two new views. From your terminal:

```
touch app/views/users/edit.html.erb
touch app/views/users/_edit_form.html.erb
```

Update the edit view like this:

app/views/users/edit.html.erb

```
<%= render "edit_form", user: @user %>
```

And then fill in the edit form partial:

app/views/users/_edit_form.html.erb

```
<%= turbo_frame_tag dom_id(user) do %>
  <%= form_with model: user do |form| %>
    <div class="flex flex-col md:flex-row items-center
px-4 py-4 sm:px-6 justify-between items-baseline">
      <div>
        <%= form.text_field :name %>
      </div>
      <div>
        <%= form.text_field :email %>
      </div>
      <div class="text-right">
        <%= form.submit 'Save', class: 'btn-primary' %>
      </div>
    </div>
  <% end %>
<% end %>
```

The key piece of code here is the Turbo Frame wrapping the form. When the **`edit`** action renders, it will return a Turbo Frame that matches the Turbo Frame we render in the **`user`** partial:

app/views/users/_user.html.erb

```
<%= turbo_frame_tag dom_id(user) do %>
```

Because each user's edit link is nested within a Turbo Frame, clicking on the edit link for a user will scope that navigation within the Turbo Frame. The **`edit`** action then returns the HTML from **`app/views/users/edit.html.erb`**.

This view contains a Turbo Frame that matches the Turbo Frame the request is scoped to, and Turbo replaces the content of the **`user`** partial with the content of the **`edit`** view.

Partial page replacement with Turbo Frames without touching Turbo Streams or Stimulus, as we are demonstrating here, is one of the most common ways to incrementally add Turbo functionality to your application. In addition to inline editing, Turbo Frames can easily power tabbed interfaces and, as we saw in a more complex example earlier in this book, search and filter UIs.

Another, non-Turbo thing to note is that the edit form only has two inputs: name and email. In the **`update`** action will transform the **`name`** value into a first and last name.

Head to the **`UsersController`** and fill in the **`update`** action:

app/controllers/users_controller.rb

```
36  def update
37    set_name
38    if @user.update(user_params.except(:name))
39      render(@user)
40    else
41      render partial: 'edit_form', locals: { user: @user
42  }
43    end
    end
```

Then update the **`users_params`** method and add a new **`set_name`** method to the **`UsersController`**:

app/controllers/users_controller.rb

```
    private

    def set_user
      @user = User.find(params[:id])
    end
```

```
54  def set_name
55    first_name, last_name = user_params[:name].split(' ',
56  2)
57    @user.first_name = first_name
58    @user.last_name = last_name
59  end
60
61  def user_params
      params.require(:user).permit(:first_name, :last_name,
    :email, :name)
    end
```

When the form submission is valid, we render the `user` partial with `render(user)`. The `user` partial has a matching Turbo Frame id, so the content of the edit form's Turbo Frame gets replaced, and the updated user information replaces the form in the user list.

When the form submission is invalid, we render the edit form again with the invalid user object.

Be sure to take note of the addition of the `name` param to the list of parameters allowed in `user_params`.

Head to **http://localhost:3000/users** and click the edit button. Change their name or email address and click the save button and see that their information updates seamlessly.

Now try editing again, remove the user's email from the input and try to save it. The change will be rejected and the user edit form will render again but we have a problem. Because we are not rendering form errors on the invalid input there is no feedback to the user about what went wrong. The form just appears to be stuck.

You may have noticed this issue on other forms in the application. Rendering errors on the input(s) that caused the error is a common pattern in web applications but none of our forms render errors inline. In a commercial application, you might address this issue by using a gem like Simple Form;

however, it is possible to automatically add form errors inline without resorting to a gem. Let's take a quick detour from Turbo to add inline form errors across the application.

To add inline errors so user's know what has gone wrong when they submit invalid information on a form, start in your terminal:

```
touch config/initializers/form_errors.rb
```

And then update `form_errors.rb` like this:

config/initializers/form_errors.rb

```
ActionView::Base.field_error_proc = proc do |html_tag,
instance|
  html_doc =
Nokogiri::HTML::DocumentFragment.parse(html_tag,
Encoding::UTF_8.to_s)
  element = html_doc.children[0]

  if element
    element.add_class('border-red-500 focus:ring-red-
600 focus:border-red-600')

    if %w[input select textarea select].include?
element.name
      instance.raw "#{html_doc.to_html} <p
class=\"text-red-500\">#{instance.error_message.join(',
').humanize}</p>"
    else
      instance.raw html_doc.to_html
    end
  else
    html_tag
  end
end
```

Here, we are overriding the **default** `ActionView::Base.field_error_proc` with our Tailwind error classes and inserting the error message under the input field.

This `field_error_proc` configuration is not well documented, but fortunately it has been blogged about for years. This particular configuration was adapted from **this excellent blog post**.

Because we placed this code in an initializer, you must reboot your Rails server before this change will take effect.

After you restart the server, we need to make one more configuration change. Tailwind purges all unused css classes automatically, but "unused" can be a little tricky. Head over to `tailwind.config.js` and update it like this:

tailwind.config.js

```
module.exports = {
  mode: 'jit',
  content: [
    "./app/**/*.html.erb",
    "./app/helpers/**/*.rb",
    "./app/javascript/**/*.js",
  ],
  safelist: [
    'border-red-500',
    'focus:border-red-600',
    'focus:ring-red-600',
    'text-red-500',
  ],
  plugins: [
    require('@tailwindcss/forms')
  ],
}
```

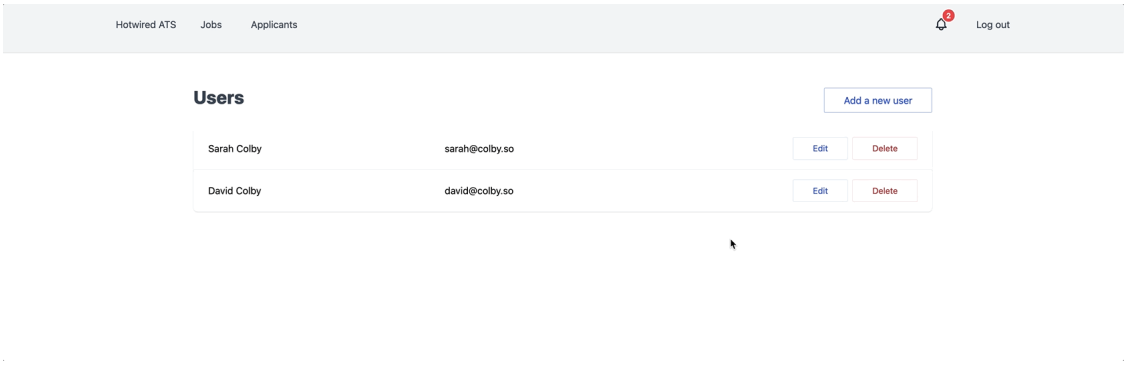
Here, we added a `safelist` value with the error classes we are using in the `form_errors` initializer. We need to do this because Tailwind's purging only scans the directories specified in the `content` array to decide which classes to keep. `config` is not in the content array and we are not using the `red-500`

and `red-600` classes anywhere else, so we specify those classes in the safest instead to instruct Tailwind not to purge them.

We could add the `config` directory to the `content` array, but it is very unusual to reference css classes in that directory, so the `safelist` option feels a bit better.

Thank you for following along with this Tailwind rabbit hole.

With Tailwind updated, we can refresh the page, edit a user and submit the form with an empty email address. If all has gone well we should see the email field highlighted along with a brief error message:



Let's finish up the CRUD actions for users by adding some familiar looking CableReady in the `UsersController`:

app/controllers/users_controller.rb

```
45  def destroy
46    @user.destroy
47    render operations: cable_car.remove(selector:
48  dom_id(@user))
    end
```

Now we can create, read, update, and delete users in the database. In a commercial application, we would put some restrictions in place to prevent users from deleting themselves or deleting all of the users in their account, but we'll trust ourselves not to do that in this application.

Add settings dropdown menu

Now we have a functional user management page but no way for users to find that page in the UI.

Let's add a new section to the authenticated nav menu, demonstrating the reusability of the `dropdown` Stimulus controller:

First, create a new partial to hold the users link:

```
touch app/views/nav/_settings_menu.html.erb
```

And then fill that partial in:

app/views/nav/_settings_menu.html.erb

```
<div data-controller="dropdown" class="relative" data-
action="click@window->dropdown#hide">
  <button class="py-4 px-1 relative border-2 border-
transparent text-gray-800 rounded-full" id="settings-
menu" data-action="click->dropdown#toggle" aria-
expanded="false">
    Settings
  </button>
  <div class="dropdown-content absolute w-48 px-2 py-2
bg-gray-100 shadow-lg rounded-md ring-1 ring-black
ring-opacity-5" data-dropdown-target="content">
    <ul class="divide-gray-200 divide-y-2" role="menu"
aria-orientation="vertical" aria-labelledby="settings-
menu">
      <li class="p-2">
        <%= link_to "Career page",
careers_account_path(current_user.account), class:
"text-blue-700 hover:text-blue-900", target: "_blank"
%>
      </li>
      <li class="p-2">
        <%= link_to "Manage Users", users_path, class:
"text-blue-700 hover:text-blue-900" %>
```

```
          </li>
        </ul>
      </div>
    </div>
```

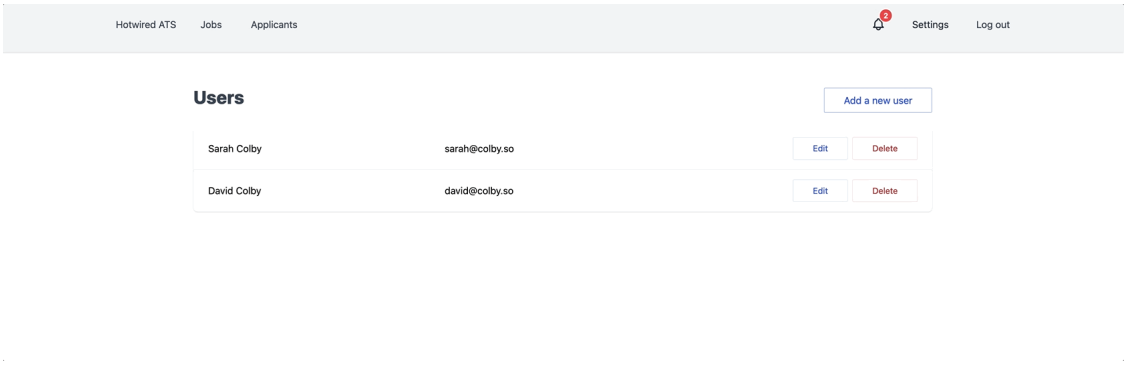
Here, we are using very similar markup to the existing `notifications` partial.

First we connect the `dropdown` controller and listen to clicks on the window. Then we add a button to show and hide the dropdown's content. In this case, instead of a list of notifications, we just render links to the users page and to the account's career page, since that doesn't fit anywhere else in the navigation menu right now.

Finally add the new `settings_menu` partial to the authenticated nav:

app/views/nav/_authenticated.html.erb

```
7 <%= render "nav/notifications", user: current_user %>
8 <%= render "nav/settings_menu" %>
```



Now that we can create new users in an account, we will wrap up this chapter by making it possible for those new users to set a password and login to their accounts.

Invite users via email

When a new user is created in an account we generate a random password for them which means there is no way for a newly created user to login except by 1) finding out that they have a user account and 2) visiting the login page and using the forgot password function to create a new password.

No real person is going to want to use our application badly enough to do all of that, so instead we are going to make it easy for them by emailing them a link they can use to “create” a user account and set a password themselves.

We will start by adding new fields to the users table. From your terminal generate a new migration:

```
rails g migration AddInviteFieldsToUsers
invite_token:string:index invited_at:datetime
accepted_invite_at:datetime invited_by:references
```

And then update the generated migration file:

db/migrate/[timestamp]_add_invite_fields_to_users.rb

```
class AddInviteFieldsToUsers <
ActiveRecord::Migration[7.0]
  def change
    add_column :users, :invite_token, :string
    add_index :users, :invite_token
    add_column :users, :invited_at, :datetime
    add_column :users, :accepted_invite_at, :datetime
    add_reference :users, :invited_by, type: :uuid,
foreign_key: { to_table: :users }
  end
end
```

The **`invite_token`** field is what we will use to find users when they visit the activation page and we’ll use the time stamps and the **`invited_by`** relationship to display useful information on the user index page.

After updating the migration file, migrate the database:

```
rails db:migrate
```

And then update the `User` model to add the new associations:

app/models/user.rb

```
belongs_to :invited_by, required: false, class_name:
'User'

has_many :invited_users, class_name: 'User',
foreign_key: 'invited_by_id', dependent: :nullify,
inverse_of: :invited_by
```

Note the `required: false` on the `belongs_to` side of the relationship — users who sign up for new accounts will not be invited by anyone so `invited_by` will be blank for those users.

Update the `create` action in the `UsersController` to populate these new fields each time a new user is added from the users page:

app/controllers/users_controller.rb

```
    def create
      @user = User.new(user_params)
      @user.account = current_user.account
      @user.password = SecureRandom.alphanumeric(24)
      @user.invited_at = Time.current
      @user.invite_token = Devise.friendly_token
      @user.invited_by = current_user

      if @user.save
        html = render_to_string(partial: 'user', locals: {
    user: @user })
        render operations: cable_car
          .prepend('#users', html: html)
          .dispatch_event(name: 'submit:success')
      else
        html = render_to_string(partial: 'form', locals: {
```

```
user: @user })
    render operations: cable_car
      .inner_html('#user-form', html: html), status:
:unprocessable_entity
  end
end
```

The `friendly_token` method is **provided by Devise** as a way to easily generate URL safe tokens, which is perfect for our needs.

Now that we have the fields we need in the database, our next step is to add a new mailer that we can use to send an email when a new user is added on the users page.

This email will contain information about who invited the user to the application and a link they can use to “activate” their new account.

First, generate a mailer from your terminal:

```
rails g mailer UserInvite invite
```

And then update that mailer in `app/mailers/user_invite_mailer.rb`:

app/mailers/user_invite_mailer.rb

```
class UserInviteMailer < ApplicationMailer
  def invite(user)
    @user = user
    @inviting_user = user.invited_by
    mail(
      to: @user.email,
      subject: "#{@user.invited_by.name} wants you to
join Hotwired ATS"
    )
  end
end
```

And then update `app/views/user_invite_mailer/invite.html.erb`:

app/views/user_invite_mailer/invite.html.erb

```
<p>
  Hi <%= @user.first_name %>,
</p>
<p>
  You've been invited to join Hotwired ATS by <%=
@inviting_user.name %>. You can activate your account
here: <%= link_to "Activate Account",
accept_invite_url(token: @user.invite_token) %>
</p>
```

Note the `accept_invite_url` method. We pass in `@user.invite_token`, which adds `token` as a parameter to the generated URL, resulting in a link that looks like `/invite?token=some_token`.

`accept_invite_url` is not defined yet. Let's define it next. First, generate a new controller from your terminal:

```
rails g controller Invites
touch app/views/invites/new.html.erb
```

And then update `config/routes.rb` to add the invite related routes:

config/routes.rb

```
resources :invites, only: %i[create update]
get 'invite', to: 'invites#new', as: 'accept_invite'
```

Here, we clean up the user-facing URL a bit by add a custom `/invite` route that points to `invites#new`.

Update the `InvitesController`:

app/controllers/invites_controller.rb

```ruby
class InvitesController < ApplicationController
  def new
    @user = User.find_by(invite_token: params[:token])
    unless params[:token].present? && @user.present?
      redirect_to root_path, error: 'Invalid invitation
code'
    end
  end

  def create
    @user = User.find_by(invite_token: params[:user]
[:token])
    if @user && @user.update(user_params)
      @user.update_columns(invite_token: nil,
accepted_invite_at: Time.current)
      sign_in(@user)
      flash[:success] = 'Invite accepted. Welcome to
Hotwired ATS!'
      redirect_to root_path
    else
      render :new
    end
  end

  private

  def user_params
    params.require(:user).permit(:first_name,
:last_name, :password)
  end
end
```

Notice that we use the `token` to look up the user in the `new` and `create` actions. The `new` action fails and boots the visitor to the `root_path` if no user is found with the provided token.

The `create` action processes the user's invite form submission and signs them into their account on a successful submission. When the user activates

their account successfully we also clear out their login token so that it cannot be used again in the future.

Next, update the new view:

app/views/invites/new.html.erb

```
<div class="flex flex-col justify-center py-12 sm:px-6
lg:px-8">
  <div class="sm:mx-auto sm:w-full sm:max-w-lg">
    <h2 class="mt-6 text-center text-3xl font-extrabold
text-gray-900">
      Welcome to Hotwired ATS
    </h2>
  </div>

  <div class="mt-8 sm:mx-auto sm:w-full sm:max-w-lg">
    <div class="bg-white py-8 px-4 shadow sm:rounded-lg
sm:px-10">
      <%= form_with(model: @user, url: invites_path,
method: :post, html: { class: "space-y-6" }) do |form|
%>
        <%= form.hidden_field :token, value:
params[:token] %>

        <div class="form-group">
          <%= form.label :first_name %>
          <div class="mt-1">
            <%= form.text_field :first_name, autofocus:
true %>
          </div>
        </div>

        <div class="form-group">
          <%= form.label :last_name %>
          <div class="mt-1">
            <%= form.text_field :last_name, autofocus:
true %>
          </div>
        </div>
```

```
        <div class="form-group">
          <%= form.label :password %>
          <div class="mt-1">
            <%= form.password_field :password,
autocomplete: "current-password" %>
          </div>
        </div>

        <div>
          <%= form.button 'Accept invite', class: "btn-
primary w-full", data: { disable_with: 'Submitting' }
%>
        </div>
      <% end %>
    </div>
  </div>
</div>
```

Users will use this form to accept invites to join an existing account in our application.

The form prefills their name and asks the user to set their own password. We include the invite token as a hidden field on the form so that we can associate the form submission with the invited user on the server.

The invite mailer we created earlier is not yet being sent. Update the **`UsersController`** to send the **`invite`** template to the new user after the save is successful.

app/controllers/users_controller.rb

```
    def create
      @user = User.new(user_params)
      @user.account = current_user.account
      @user.password = SecureRandom.alphanumeric(24)
      @user.invited_at = Time.current
      @user.invite_token = Devise.friendly_token
```

```
  @user.invited_by = current_user

  if @user.save
    UserInviteMailer.invite(@user).deliver_later
    html = render_to_string(partial: 'user', locals: {
user: @user })
    render operations: cable_car
      .prepend('#users', html: html)
      .dispatch_event(name: 'submit:success')
  else
    html = render_to_string(partial: 'form', locals: {
user: @user })
    render operations: cable_car
      .inner_html('#user-form', html: html), status:
:unprocessable_entity
  end
end
```

Note that we are queuing up this email in the controller action instead of using a callback in the `User` model. Only users created from the `create` action in the users controller should receive this email, so triggering the email delivery in the controller feels a bit more clear.

User invite emails are now ready to be sent to new users added from the users page. Head to the users page now and create a new user. If all has gone, you should see the `invite` email template sent to the new user after the `create` action succeeds.

Grab the invite link from the logs (or from the `letter_opener_web` interface, if you are using it) and see that you can use that link to activate the new user and login as them.

Reinviting users

Let's wrap up this chapter by adding invite information to the users list and adding a small feature to allow users to send new invite emails to users that

have not yet activated their account.

First, we will update the user partial:

app/views/users/_user.html.erb

```
<%= turbo_frame_tag dom_id(user) do %>
  <div class="flex flex-col md:flex-row items-center
px-4 py-4 sm:px-6 justify-between">
    <div class="w-1/3">
      <%= user.name %>
      <div class="flex space-x-2 text-gray-500 text-
sm">
        <div>
          <%= user_invite_info(user) %>
        </div>
      </div>
    </div>
    <div class="w-1/3">
      <%= user.email %>
    </div>
    <div class="flex justify-end w-1/3">
      <% if user.invited_by.present? &&
user.accepted_invite_at.blank? %>
        <%= button_to "Resend invite",
invite_path(user), method: :patch, class: "btn border
border-blue-200 hover:bg-blue-200 text-sm text-blue-700
mr-2", remote: true %>
      <% else %>
        <%= link_to "Edit", edit_user_path(user),
class: "btn border border-blue-200 hover:bg-blue-200
text-sm text-blue-700 mr-2" %>
      <% end %>
      <%= link_to "Delete", user_path(user), method:
:delete, class: "btn border border-red-200 hover:bg-
red-100 text-sm text-red-700", data: { confirm: "Are
you sure?" } %>
    </div>
```

```
      </div>
    <% end %>
```

Here, we added a call to a helper method, `user_invite_info` which is not yet defined and added a simple conditional to display a "Resend invite" button for users that have not activated their accounts yet.

Now, let's define the `user_invite_info` method in `UsersHelper`:

app/helpers/users_helper.rb

```
module UsersHelper
  def user_invite_info(user)
    return "Signed up on #{l(user.created_at.to_date,
format: :long)}" unless user.invited_by.present?

    if user.accepted_invite_at.present?
      "Signed up on #
{l(user.accepted_invite_at.to_date, format: :long)}"
    else
      "Invited on #{l(user.invited_at.to_date, format:
:long)}"
    end
  end
end
```

For users that have already activated their accounts we display the date they activated. Otherwise, we display the date they were invited.

Head to `InvitesController` to add the `update` action that the "Resend invite" button points to:

app/controllers/invites_controller.rb

```
  def update
    @user = User.find(params[:id])
    @user.reset_invite!(current_user)
    UserInviteMailer.invite(@user).deliver_later
```

```
    flash_html = render_to_string(partial:
'shared/flash', locals: { level: :success, content:
"Resent invite to #{@user.name}" })
    render operations: cable_car
      .inner_html('#flash-container', html: flash_html)
  end
```

Here, we call a `reset_invite!` method and then send the invite email to the user again before using CableReady operations to inform the user that the invite has been sent.

`reset_invite!` has not been defined yet, so we will finish up this chapter by heading to the `User` model to define it:

app/models/user.rb

```
def reset_invite!(inviting_user)
  update(invited_at: Time.current, invited_by:
inviting_user)
end
```

Simple ruby code here, updating the user's invite information so that it can be shown correctly in the email and on the user's index page.

And with that change in place, head to the users page again and see that users that have been invited and have not yet accepted the invite now have a Resend invite button displayed. Click it and see the invite email gets sent to them again.

That is all for this chapter! You are nearing the end of this book. We have two chapters of work left.

As usual, this is a great place to pause, commit your changes, and take a break to reflect before moving on.

If you are feeling adventurous, you might spend a little more time with the `update` action we just added for resetting user invites.

Right now, we do not update the user's invite information in the view when the `update` action runs. The "Invited on..." text should be updated without a page turn but it is not. One extra CableReady operation should be enough to get that text updated without requiring a page turn. Alternatively, this could be an opportunity to explore `updates_for` or Turbo Stream model broadcasts again.

In the next chapter, we will finally fill in the long-neglected dashboard, using StimulusReflex and ApexCharts to build two filterable charts on the dashboard.

To see the full set of changes in this chapter, review **this pull request** on Github.