Building Minibird in Ruby on Rails

You can use this document as a guideline to build a Rails application. These steps will help you set up Minibird, a tiny invoicing app. Feel free to do things the way you like to, and ask questions if something's up!

Firing up the Rails application

- Download (or clone) the application code from https://github.com/jeroenmoneybird/workshop.
- Open a terminal window, go to the directory of the Rails application.
- Run `bin/setup`. This installs missing gems and creates the sqlite database.
- Run `bin/rails server` and wait until it's done loading.
- Go to http://localhost:3000/ to see the app.
- Just leave the server running while continuing. Rails reloads any changes you make to the code!

Creating Invoices

The easiest way to get started is to use the scaffolding functionality in Rails. Of course you can always skip this if you want a real challenge, but it's recommended to start this way.

- Generate a scaffold for the `Invoice` model:
  - `bin/rails generate scaffold invoice invoice_number:string invoice_date:date`
- Take a look at the files that were generated. What's inside the controller, views and model?
- Run the database migrations using `bin/rails db:migrate`
- Find the commented line starting with `root` in `config/routes.rb` and change it to:

    - `root “invoices#index”`
- Now reload the browser and see what happens!
- You can create, update and delete invoices, with just a single command!

Invoice Lines

An empty invoice isn’t very useful. It needs some invoice lines to tell the customer what has been charged, and how much it costs.

Generate a scaffold for the `InvoiceLine` model. It needs the following attributes:

```
invoice:references price:decimal quantity:integer
description:string
```

The `invoice:references` will add an `invoice_id` column to the `invoice_lines` table. It *references* the invoice. Rails even adds the foreign key for you!

Run the database migrations and go to http://localhost:3000/invoice_lines and try to add an invoice line.

Nesting Resources

You might be wondering what the Invoice field is for, you might even have seen some errors trying to save an invoice line.

The problem is that an invoice line could not really exist on its own. It *belongs to* an invoice. You could try to add an existing invoice ID in the Invoice field, that would actually work! But it’s not that convenient.

We should actually ‘wrap’ the invoice around these invoice lines so that you cannot even attempt to create an invoice line without an invoice. Rails offers a handy mechanism for this, called nested resources.
First, take a look at the routes as they are now, using `bin/rails routes`. Then, move the `invoice_lines` resource into `invoices`, as such:

```
resource :invoices do
  resources :invoice_lines, as: :lines
end
```

Don’t worry about the `as: :lines`. It’s just something to make the names of the route helper methods a bit more logical.

Now take another look at the routes. You’ll see that the URLs for invoice lines all include an `invoice_id` parameter now.

Linking to invoice lines from invoice

Of course an invoice needs to have an action to add a new invoice line to it.

The show action of an invoice should provide this link. Inspect the output of `bin/rails routes` again and add the link in `app/views/invoices/show.html.erb`. There are already some other links in there to use as an example. Be sure to pass `@invoice` as an argument to the route helper method!

Reload the browser and see that the link has been added. Click it and see what happens. Also note that the URL in your browser now includes the invoice’s ID.

Automatically selecting the invoice

The Invoice field is still there in the invoice line form. This is still not very convenient. But now that the `invoice_id` is included in the URL, it can be fetched from `params` to automatically fill in the Invoice field!
Open `app/models/invoice.rb` and `app/models/invoice_line.rb`. An invoice line belongs to an invoice, so it has a `belongs_to` relationship which Rails automatically added for you. But invoice *has many* invoice lines, however that end of the relationship is still missing.

Luckily, this doesn't require a change to the database. Simply add this line to the `Invoice` class to set up the other end of the relationship:

```
  has_many :invoice_lines
```

Now go back to `app/views/invoices/show.html.erb`. This is where the invoice lines should be shown as well. Find the render call that's there, and add this line underneath it:

```
 <%= render @invoice.invoice_lines %>
```

Now, for each invoice line present in the invoice, it will render the invoice line partial. Go back to the browser and open an invoice. Note that since it has no invoice lines yet, nothing is shown.

The final step is to make sure that an invoice line has the Invoice field prefilled. For that, the `InvoiceLinesController` needs to be changed.

Add this method above the `set_invoice_line` method:

```
  def set_invoice
    @invoice = Invoice.find(params[:invoice_id])
  end
```

Then add this line before the other `before_action`:

```
  before_action :set_invoice
```

`before_action` is a callback that will call the referenced method before the controller action method is called. So before `new` is called, the `@invoice` instance variable has been set to the `Invoice` that's referred to in the URL.

To actually fill the Invoice field with the invoice ID, you need to replace the 2 occurrences of `InvoiceLine.new` with `@invoice.invoice_lines.build`. This takes the `Invoice` instance, and then calls `build` on its `invoice_lines` relationship. This initializes an `InvoiceLine` model instance, but doesn't save it to the database yet. It does fill the `invoice_id` attribute, which is what we need!

Go back to the browser and try to add an invoice line to an invoice again. Note that now, the Invoice field has been prefilled with the ID of the invoice! Try adding one.

Sadly, that runs into some kind of error. The controller tries to redirect back to the show page of an invoice line. But since an invoice line is a nested resource, it always needs to have the ID of the invoice in the URL too, which is missing now.

It would make more sense if the `InvoiceLinesController` always redirects back to the invoice instead of the invoice line. That way, you get a nicer workflow if you want to add more lines to one invoice.

Rewrite each of the redirects so that it always redirects to `@invoice` instead. Now try it out again, this time it works!

Lastly, you might as well remove the Invoice field from the form now. It's now unused since we're building the invoice line from the invoice instead. Can you figure out where the invoice line form is and remove the field?

Cleaning up InvoiceLinesController

Because we've used scaffolding for `InvoiceLinesController`, we got a complete controller with all actions, even though some aren't actually used anymore now. Can you figure out which ones they are and clean up the controller action and views?

Adding model validations

Did you already try to save an invoice or invoice line with some fields left blank? You can store incomplete data now… You could add validations to both models to prevent empty fields from being added.

Check out the Rails documentation if you need some examples.

Getting a better overview

Once you've added a couple of invoice lines, things start to get out of hand quickly in the invoice view. It's hard to see where one invoice line begins, and the other one ends.

Maybe it's a good idea to use a `<table>` for the invoice lines to get a better overview of the invoice.

Invoice line totals

Now that you've got a better layout of your invoice lines, maybe it's a good idea to show the total price of each invoice line. The total of an invoice line is its quantity, multiplied by the price.

Try defining a method `total_price` in `InvoiceLine` which calculates the total price of the line. You can then display this number for each invoice line in the views.

Invoice total

Calculating the invoice line totals is nice, but it doesn't show exactly what the customer has to pay for the invoice. Try adding the total price of the whole invoice in the invoice show view as well.

Ensuring invoice number correctness

Invoice numbers are always incrementing. So if your previous invoice number was 2022-001, then your next invoice should have a number like 2022-002.

If the invoice number has to be filled out by the user every time, this could easily go wrong, and you'd end up with duplicate invoice numbers. And the tax authorities won't be so happy with that…

First, it's important to ensure that no duplicate invoice number can be present in the database. For this, you need a unique index.

Execute `bin/rails generate migration add_unique_index_invoices_on_invoice_number` in the terminal. This will create a new migration file, containing a single `change` method. As you know, there's a migrate command, and a rollback command. But a lot of the Rails migration commands can be rolled back automatically, using Rails magic! So for most migrations, just implementing the `change` method will do. Check out the Rails documentation if you want to know more about migrations.

Add this line to the method body:

```
  add_index :invoices, :invoice_number, unique: true
```

Then run the migrations. This will add the unique index, ensuring that no duplicates enter the database. This *might* fail if you have the same invoice numbers in your database already, so in that case just fix the offending invoices and try again.

If the migration succeeded, try adding a duplicate invoice number. You'll get an error telling you that a database constraint has failed.

The next step is to automatically fill in the invoice number. In the new action of your `InvoicesController`, you can slightly change the way the new `Invoice` is initialized:

```
def new
  @invoice = Invoice.new(
    invoice_number: Invoice.maximum(:invoice_number).next
  )
end
```

`maximum` is an ActiveRecord method fetching the maximum invoice number from the database (documentation). `next` returns the next possible string value (documentation). Just a few simple pieces of Rails and Ruby code combined can be quite powerful.

Try adding a new invoice. Now, the invoice number is automatically filled for you.

You could do the same with the `invoice_date`, making today's date the default value. That should automate things a bit!

Better price formatting

The prices don't actually look like prices yet, more like numbers now. Fortunately, Rails has a lot of useful helper methods. One of them can convert the boring number to a formatted monetary amount with currency!

Check out the view helpers section of the Rails documentation. Can you figure out which one of the helper methods to use here?

Next steps

Whoa, you made it all the way down here? If you did, try thinking of more improvements you could make to the code, or to the views.

Or if you want a more thorough introduction into Rails, you could try building a blogging app from scratch in the Getting Started with Rails guide. It provides a lot more insights into the Rails functionality.