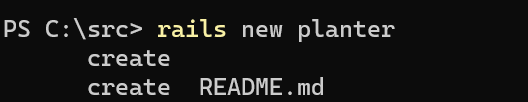
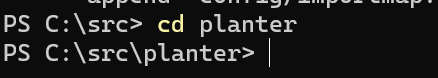
**Planter DEMO:**

1. **All we need to get started is to pick a name for our new Rails application.** I'm going to call this one planter because it's going to be a planner for plants, so I type rails new planter, and press Enter. The Rails application creation process starts. You're going to see a lot of things installing. First, Rails will check if you have all the necessary Ruby dependencies that Rails needs and install them if needed, then it will move onto front‑end JavaScript dependencies. As I mentioned earlier, Rails uses yarn for its JavaScript dependency packaging. The last thing Rails will install is Webpacker, which is used to bundle up all the JavaScript dependencies. 
2. Rails created a new directory for your apps, so you need to move into that directory with a cd, or change directory, command, so we **type cd planter.**



1. **So let me open my text editor to show you the basic structure of the app.** At the very top, we have an app directory, which will contain most of our code for the application. You can recognize here three familiar things, models, views, and controllers. But right now, this is just a skeleton, so we don't actually have any code here, aside from some Ruby‑based classes that can be used to share behavior later.
2. **Rails Scaffolding**

So it's time to do some scaffolding. Rails has a powerful generate command to create all the basic structure of models, controllers, but also full resources and scaffolds, which include all the possible initial setup you might need for a new application model or concept. We're going to need people in our application, so let's make a person scaffold and give it an email and a name.



* We've got a new migration for the people table, and a person model, a people controller, all sorts of views for all the possible CRUD actions, and even CSS style sheets.

1. **Rails Server**

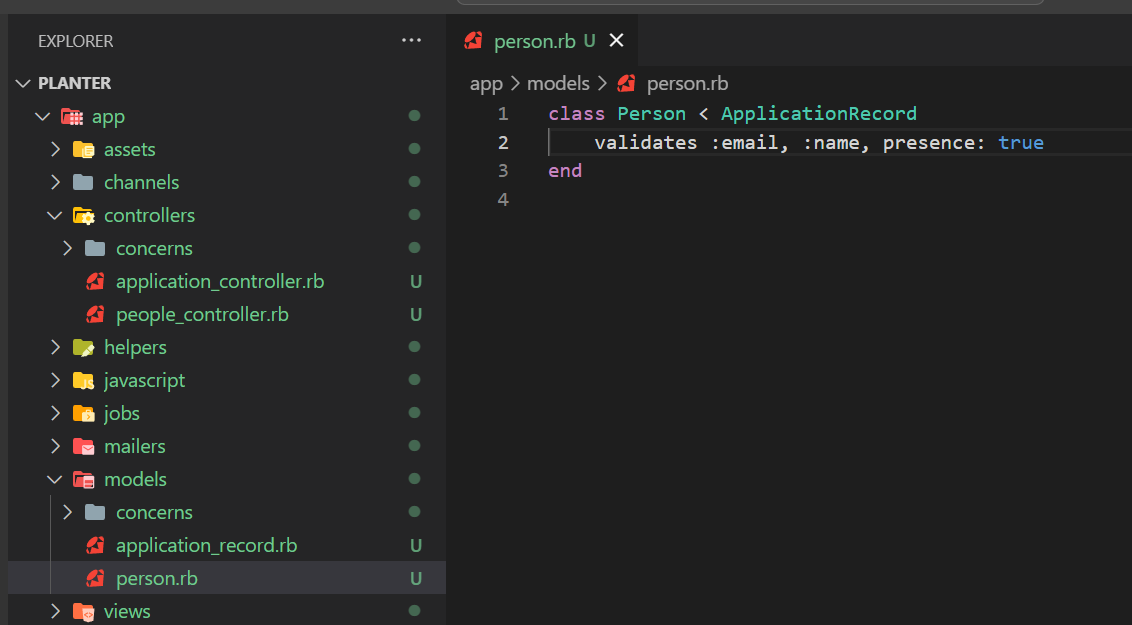
* While it's nice that Rails creates all these files for us, it's even better to see what that means in practice. Yet another Rails command, arguably the most useful, is rails server. It boots the Rails application on your machine and starts displaying some logging information about the running server.



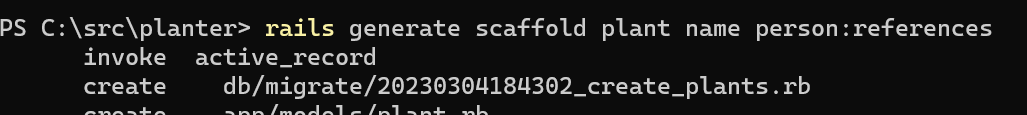
* When we visit the URL that was listed, we run into an error with a lot of red, but this actually is a nice Rails feature. It's reminding us that since we prepared a new database table for people, we probably want to ensure that it gets created in our database. We don't even have to stop the server to run this database migration. We can just hit the Run pending migrations button, and Rails will run it for us, and there we go. We're on Rails. If we go to the /people path, we can find the index page that Rails created for the scaffold and click on the New Person link to access the person creation form.

1. **Feedback Cycles**

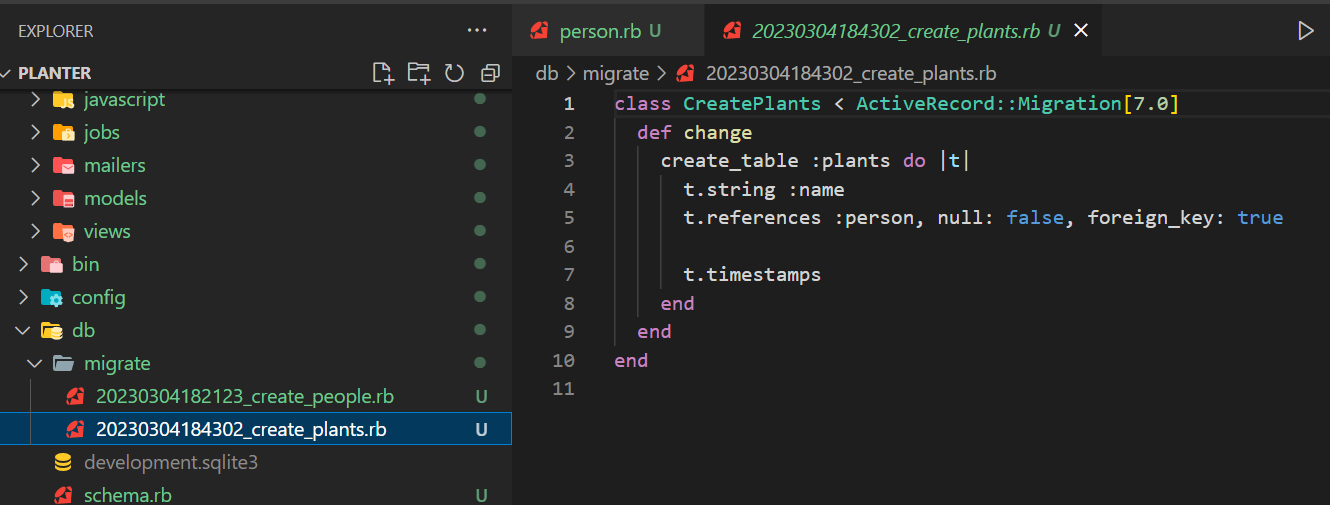
* Now that we have a scaffold and the Rail server running, let's look at the feedback cycles involved in developing a Rails application. If we go back to our People index page, we can see that we're missing an email for this person I just created. So maybe that's something we should require. We can do that using the validates method that we saw when looking at Active Record earlier. We're now validating both the email and the names. So if I go back to edit my person record, we should encounter an error when I try to update.



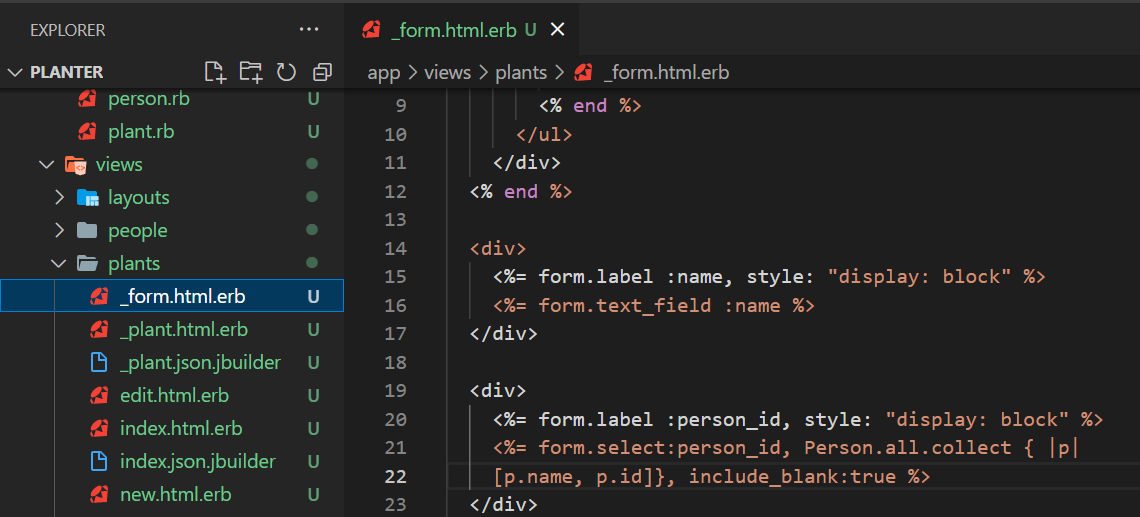
* I moved into a different terminal tab, so we can keep typing Rails commands in this new tab. In it, we're going to create a new scaffold. This time, it's going to be for a plant model with a name attribute and a reference to a person.



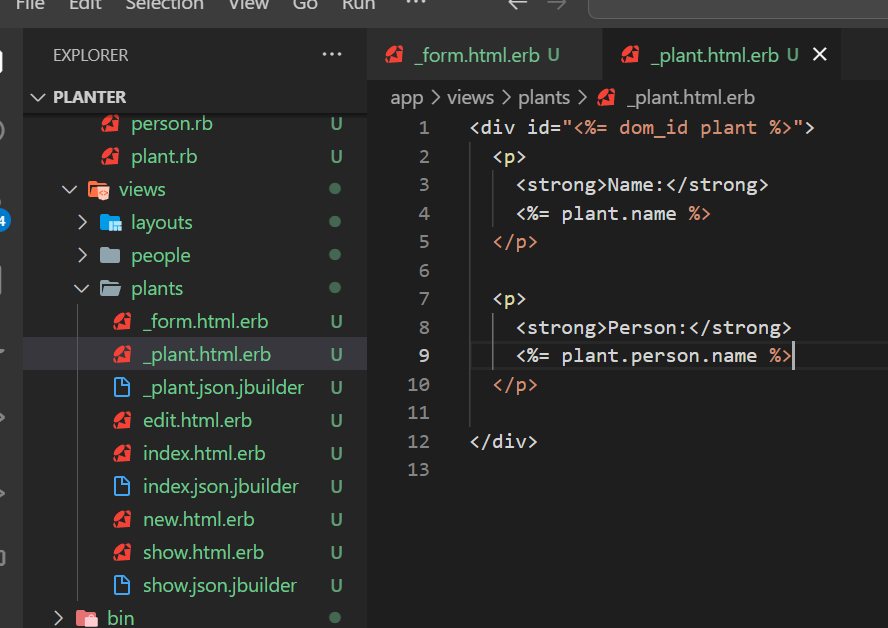
* Migrations live in the **db/migrate directory** and look like this. You can see the string method and the table block argument we saw when discussing Active Record. But there's a new method called references, which takes a person symbol and two options, null: false and foreign\_key: true. Null: false means we require all plants to always have a reference to a person. Foreign\_key: true means we're also creating a foreign key constraint on the person ID that will be created in the plants table. This means we can't have a person ID in there that doesn't also exist on the person table.



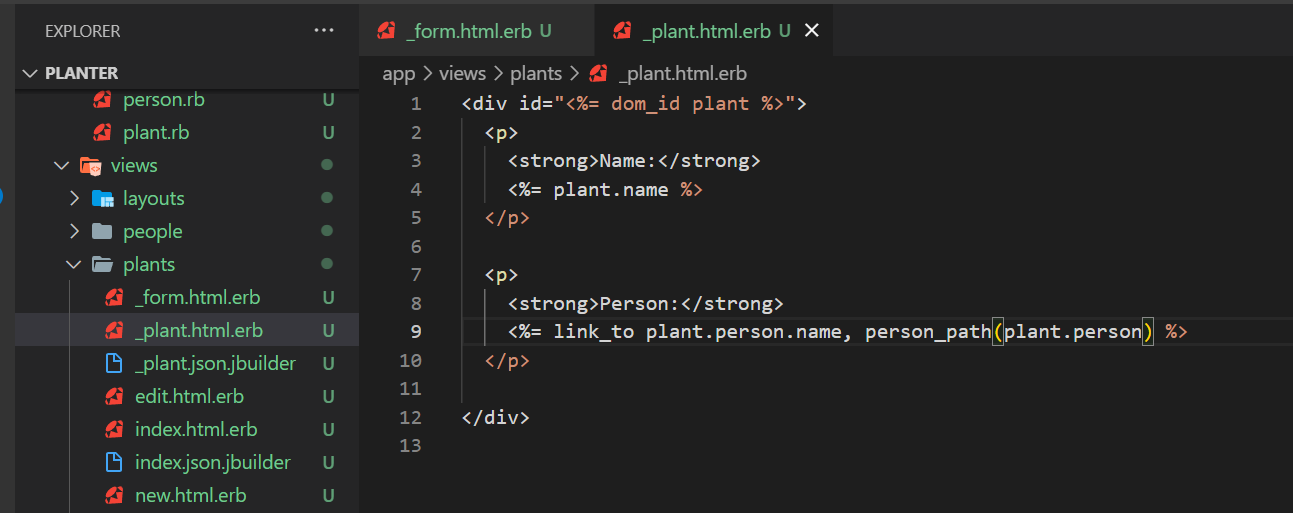
* You can give this select method a model attribute name like our person ID, and the second argument needs to return an array of two elements, the option's name and its value. In our case, the value would be the person ID, which will be sent to our Rails controller when we submit the form. Now when we go back to the plant form, the select element shows my name as one of the options when we click on it, and I can successfully create my tomato plant.



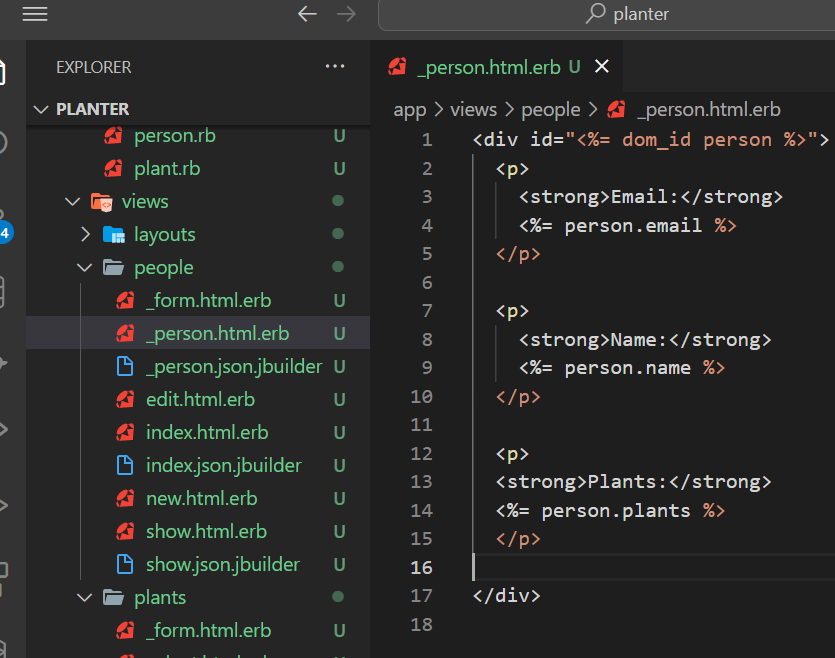
* Lets change person id to person name



* And also lets add the link to Person

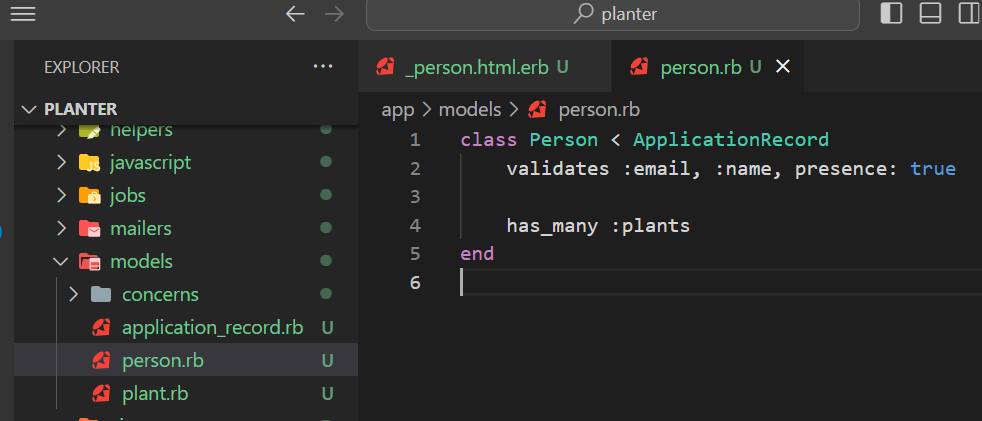


* The Plants index page lists all the plants from all people, so it might get large and hard to read as usage grows with multiple people. So, let's add a new paragraph to our People show view, and use the erb output syntax with the equals sign, to print out a given person's plants. When we reload, we get an error, a NoMethodError. It's the first time we encounter it, but it's a fairly common error in Ruby.

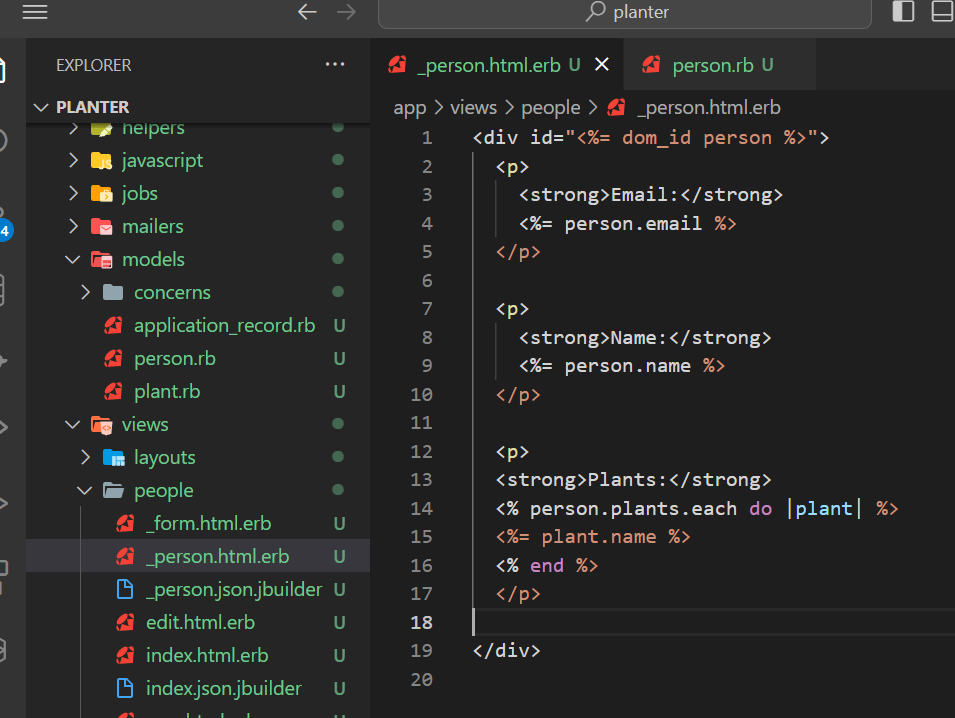


1. **Associations in Practice**

* Since the person\_id foreign key is defined on the plants table, as we can confirm inside of the generated schema that RB produced when migrations are run, what we're missing is a call to a has\_many :plants class method inside of our person model.

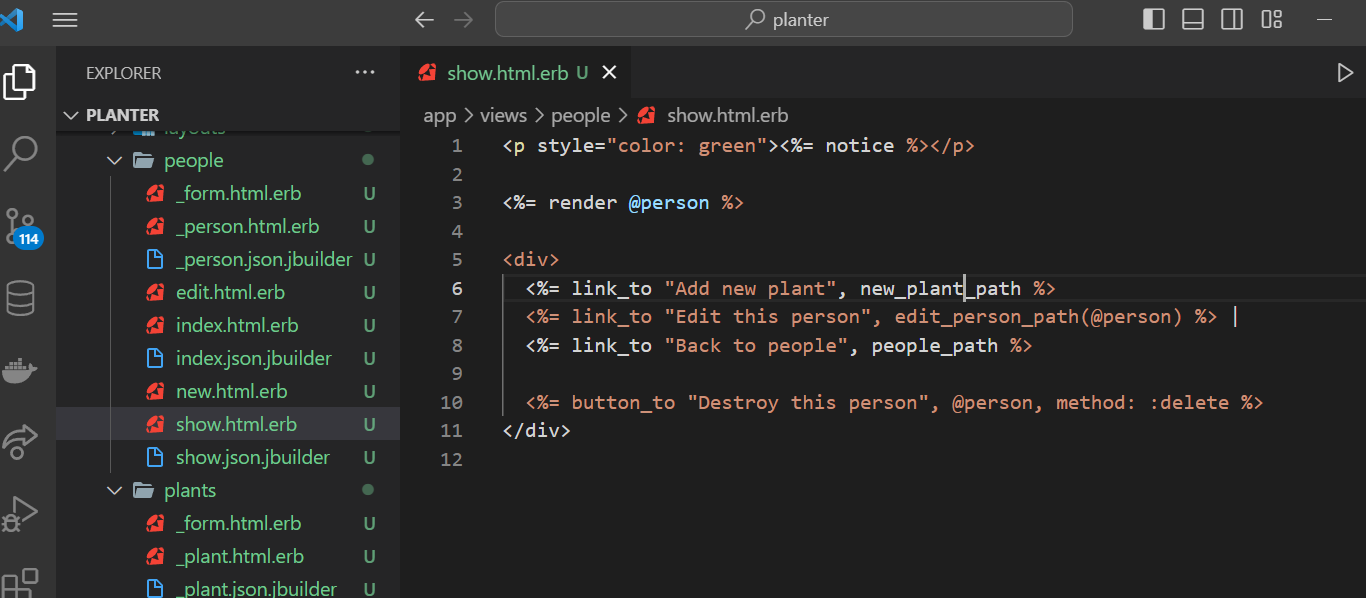
****

* Also need to include to \_person.html.erb for/each loop

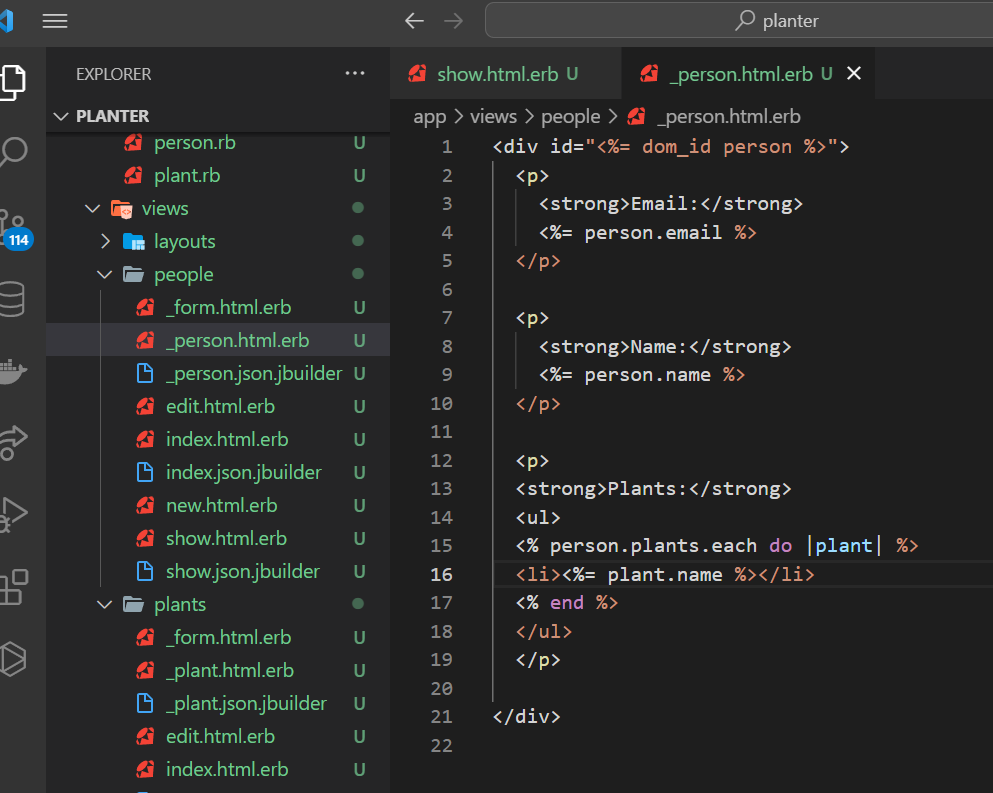


1. **Routes, Links, and Parameters**

* **Add route to New Plant to Person page**

****

* **Lets show each plant for Person**

****