Today we're going to learn Rails!

Rails is an MVC framework for building large web applications

What is a framework?

- A set of code libraries that help you achieve a certain purpose
- In Rails, the purpose is building large web apps

What is MVC?

- A common way of **architecting** an app
- The **architecture** is how its code is structured:
  - What do I name my files and folders?
  - Which files should I put my code in?
  - Which folders should I put my files in?
  - What should I name the classes in my files?
  - Etc.
- MVC stands for Model View Controller.
- We're going to talk about all this in more detail
- Learning Rails will help you learn any other MVC framework

Agenda

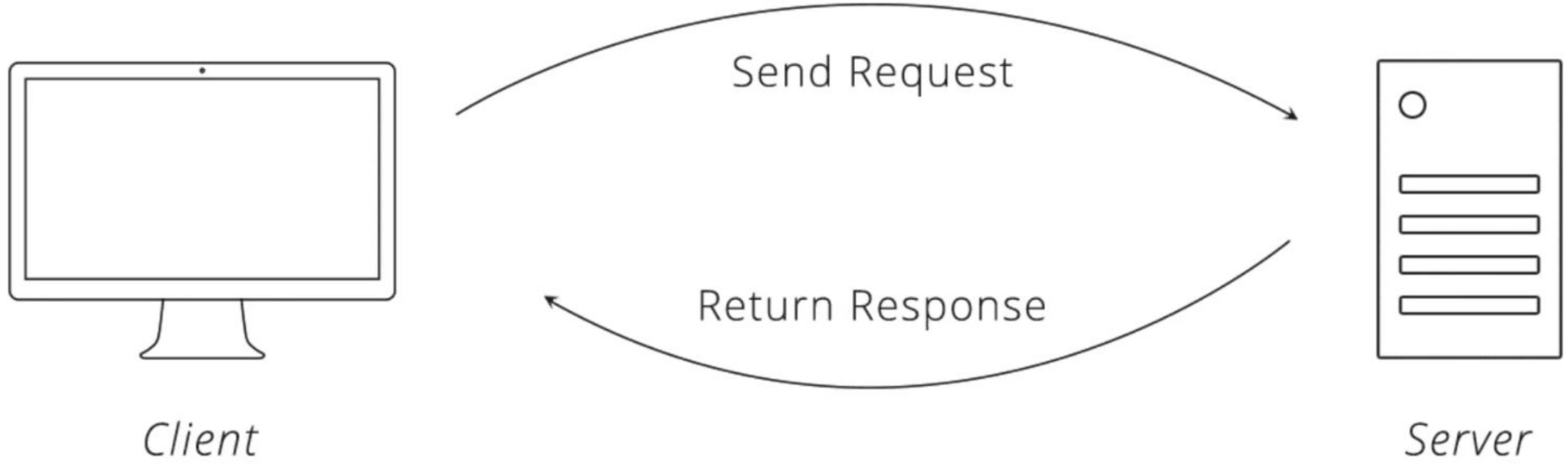
1. Quick review of previous concepts
2. What is Rails?
3. What is MVC?
4. Getting started with Rails
5. Rails demo

Everything has been building up to today

We've taught you key concepts you need to understand Rails:

1. Request/response cycle
2. Persisting data in a database
3. CRUD
4. Dynamic HTML
5. Forms

Request/response is how browsers and servers communicate



- You’ve built apps where the server is Sinatra.
- Now we’re going to build apps where the server is Rails

A request includes a type and a URL

In your Sinatra apps, you listed all the request types and URLs that your server would respond to

```
get '/contacts' do
  @contacts = Contact.all
  erb :index
end
```

```
get '/contacts/:id' do
  @contact = Contact.find(params[:id])    ])
  erb :show
end
```

We do something very similar in Rails, where we list out all the request types and URLs that the Rails server will respond to

Databases allow you to persist data

- When you restart a program, you lose the data
- If you save the data to a database, it will still be available in the database even if you restart the program
- Your Sinatra CRM app used a database to persist data
- Rails apps use a database to persist data as well

CRUD = the 4 functions of data

In any web application with data, you usually want to give your users - or yourself - the ability to CRUD records, like in your Sinatra CRM app.

Rails apps allow you to quickly build CRUD functionality.

ERB gives you the power of dynamic HTML

- In a web application, you want the HTML your users see to change when the data changes
- For example, if this in our CRM index view:

  ```
  <% @contacts.each do |contact| %>
    <li><%= contact.full_name %></li>
  <% end %>
  ```

  Then it will render more `<li>` elements if there are more contacts.
- Rails supports dynamic HTML using ERB as well

Forms allow your user to send you data

- When a user fills out a form and then submits it, the data they entered is placed into the `params` hash as Sinatra's gift to you

```
post '/contacts' do
  Contact.create(first_name: params[:first_name], last_name: params[:last_name],
                 email: params[:email], note: params[:note])
  redirect to('/')
end
```

- Rails gives you a `params` hash gift as well

All these concepts are used in Rails!

But first, a note about depth vs. breadth

- A depth-first learner is someone who wants to fully understand a concept before moving onto the next one
- Rails can be frustrating if you're a depth-first learner
- That's because there are layers upon layers of code behind even the smallest of things in Rails
- If you take a detour to try to understand how a particular method works, you might fall behind

Think of Rails like a car

- Focus on learning how to drive the car, not how the engine/wiring works
- You can get a lot of benefit out of a car even if you don’t know how it works
- Once you can drive the car, it’ll be easier for you to learn how it works
- As you start to build more complicated features, you’ll get chances to dig into the internals of Rails

Now onto Rails!

Ruby vs. Rails

Ruby

- Language
- Created in 1995

Rails

- Framework
- Created in 2004
- AKA Ruby on Rails

A brief history of Rails

- Basecamp was the first Rails app
- Rails was created by David Heinemeier Hansson, or DHH
- Realized he was doing the same thing over and over again when building web apps
- Decided to write some code that would do these things for him. Called it Rails
- Built Basecamp using Rails in 2004
- Open-sourced Rails
- Gave a viral talk in 2005

instacart

airbnb

shopify

There is a “Rails Way” to do almost everything related to a web application

- What do I name my files, folders, classes, tables, column names, methods?
    - Camel case? Snake case?
    - Plural? Singular?
- Where do I put my code?
- How do I test my app?
- How do I run my server?

Convention over configuration saves you time

- Following the Rails way = following Rails conventions
- Following conventions means less configuring to do
- This frees you up to focus on more interesting problems

More on the Rails Way

1. Keep logic separate from the UI
2. Test, test, test
3. Keep code DRY
4. Keep code modular

Keep logic separate from the UI

- Before Rails, it was common to have SQL statements and HTML markup in one long file
- Separating the code that fetches the data from the UI code into 2 files gives you 2 shorter files that are easier to understand
- It also allows you to reuse code:
    - the data fetching code could be used by another view
    - the UI code could be used to display a different set of data
- This is an MVC concept we'll talk about soon

MVC

MVC is an architectural pattern

- Invented in 1979, used for desktop, phone and web applications
- Popularized for the web by Rails
- Other MVC frameworks: Laravel (PHP), Django (Python)

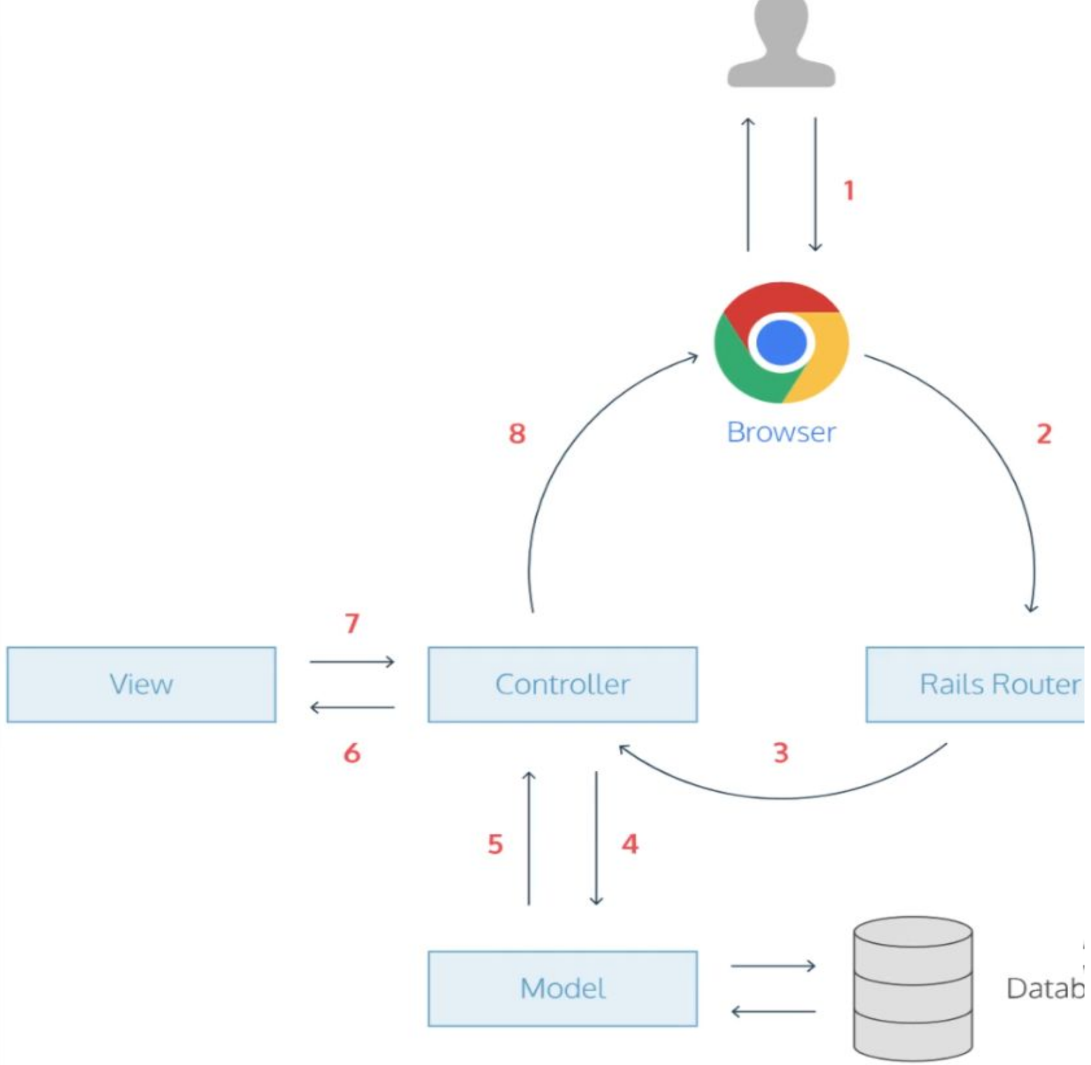
MVC allows for separation of concerns

- In MVC, we separate our code into 3 main categories: Model, View, Controller
- All UI code is in the View. No business logic code is in the View
- All business logic code is in the Model
- The controller figures out which Model code and View code is relevant t each request
- The controller
  - asks the model to execute the relevant business logic,
  - then asks the View to render the relevant template using the data from the model

Let's talk about what MVC looks like inside of a Rails server

2 is the request. 8 is the response.

You're familiar with models, databases and views already.



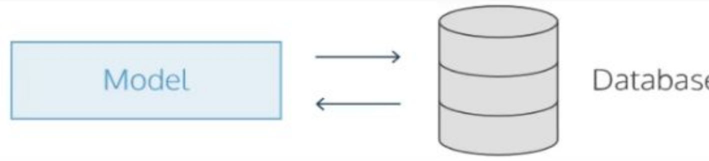
You’ve seen models in your Sinatra apps

Here’s the Contact model from your CRM app

When you execute `Contact.all`, the Contact class executes a SQL statement:

```
select * from contacts;
```

It then converts the result from the database to a collection of Ruby objects



```ruby
gem 'activerecord', '=4.2.10'
require 'active_record'
require 'mini_record'

ActiveRecord::Base.establish_connection(adapter: 'sqlite3
database: 'crm.sqlite3')

class Contact < ActiveRecord::Base
  field :first_name, as: :string
  field :last_name, as: :string
  field :email, as: :string
  field :note, as: :text
end

Contact.auto_upgrade!
```

```ruby
get '/contacts' do
  @all_contacts = Contact.all

  erb :contacts
end
```

You’ve seen views in your Sinatra apps

Sinatra passes the `@contacts` variable from server.rb to the view

The view uses this variable to build dynamic HTML - HTML that is different depending on what’s in the `@contacts` variable.

server.rb

```
get '/contacts' do
  @contacts = Contact.all
  erb :index
end
```

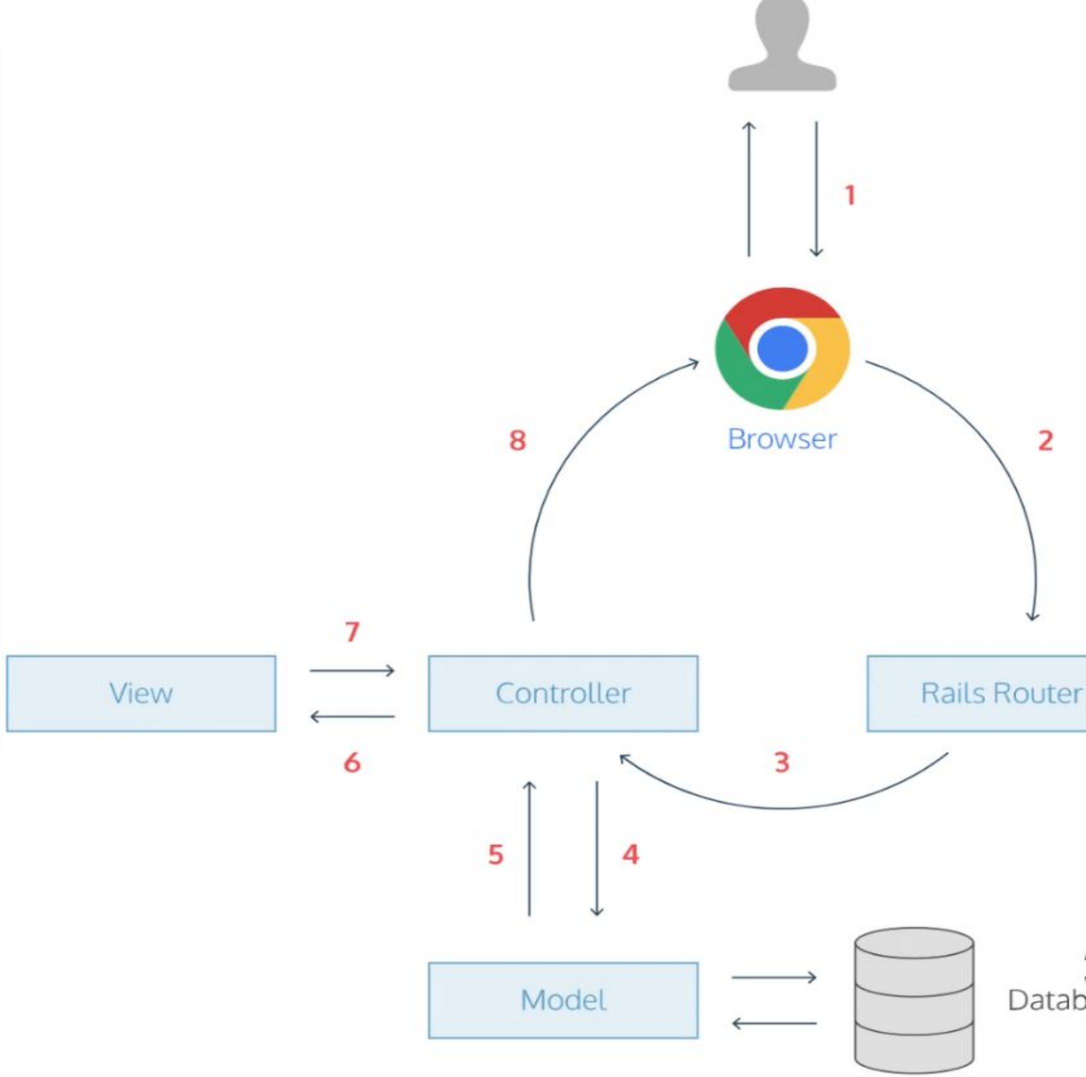
index.erb

```
<% @contacts.each do |contact| %>
  <div>
    <h2>
      <%= contact.full_name %> (<%= contact.email %>)
    </h2>
    <a href="/contacts/<%= contact.id %>">View Contact</a>
  </div>
<% end %>
```

So what does MVC look like inside of a Rails server?

1. You type in a URL
2. The browser sends a request to Rails. It goes to the Rails Router

The Router is like a receptionist in an office building. It forwards the request to the department that can handle it.



You’ve already seen router-like code in Sinatra

Here we have a Sinatra server that accepts two request:

- GET ‘/contacts’
- GET ‘/contacts/:id’

The code that responds to the request is inside the `do` blocks.

In Rails, unlike Sinatra, the code that responds to a request is separate from the code that lists the requests we accept.

```ruby
get '/contacts' do
  @contacts = Contact.all
  erb :index
end

get '/contacts/:id' do
  @contact = Contact.find(params[:id])
  erb :show
end
```

A Rails Router is a map of requests and controller-method pairs that respond to the requests

A controller method is also called an action

routes.rb

```
get '/contacts' => 'contacts#index'

get '/contacts/:id' => 'contacts#show'
```

When a request comes in, the router looks it up in its map, and forwards the request to the relevant controller

- GET '/contacts' is forwarded to:
  - Controller: contacts
  - Method: index
- GET '/contacts/2' (for example) is forwarded to:
  - Controller: contacts
  - Method: show

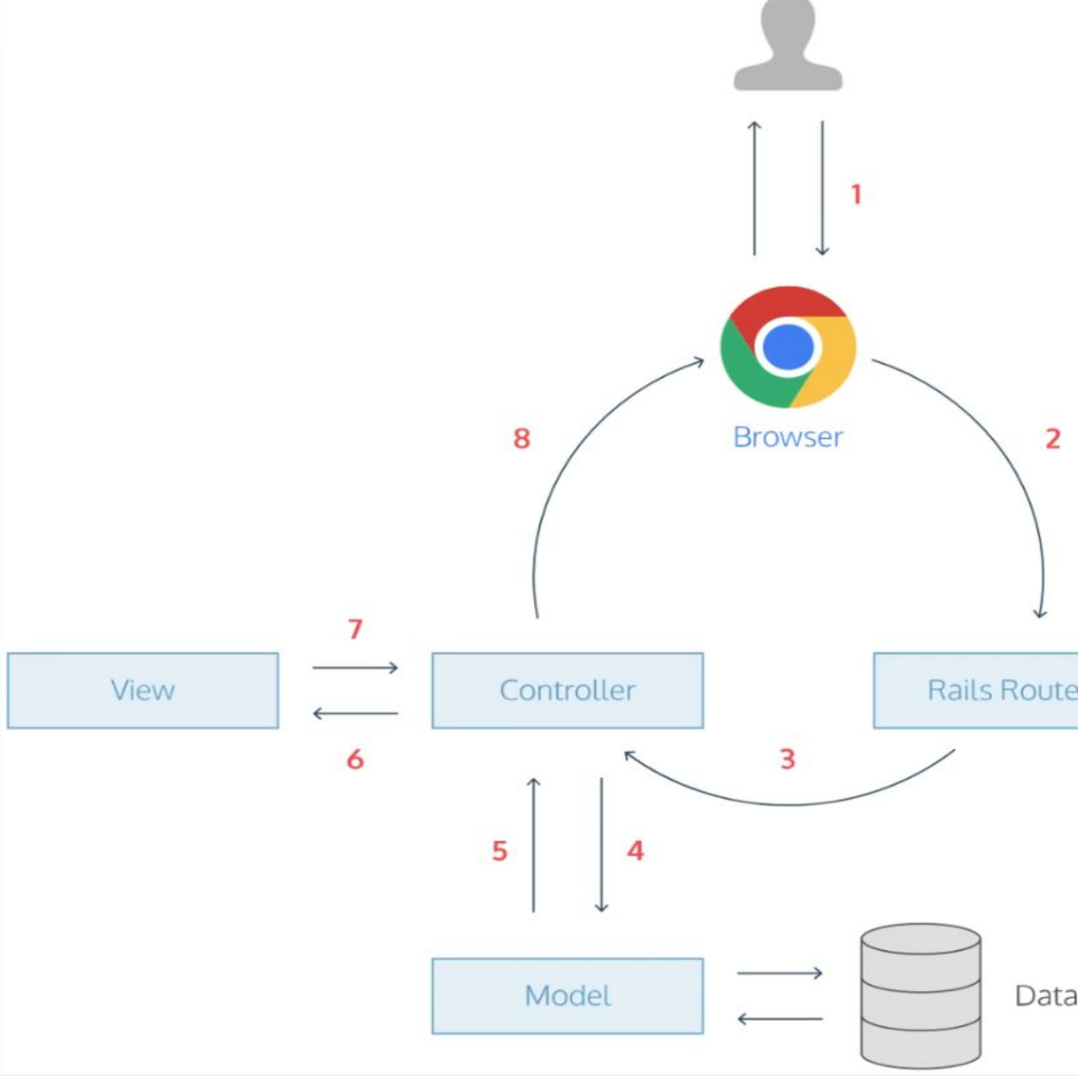
So in step 3, the request is forwarded to the relevant method in the relevant controller

GET '/contacts' is forwarded to:

- Controller: contacts
- Method: index

GET '/contacts/2' (for example) is forwarded to:

- Controller: contacts
- Method: show



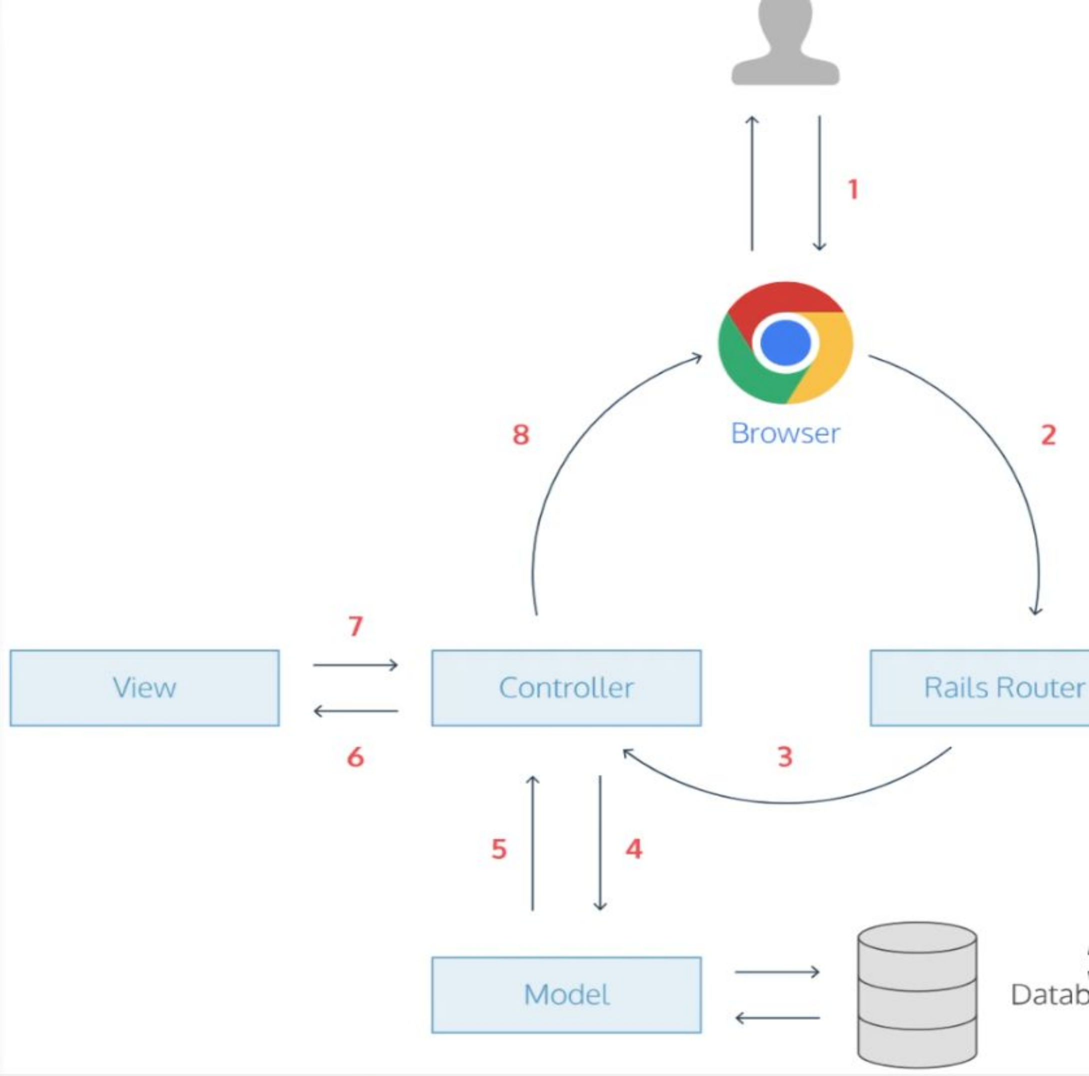
The controller is like a manager. It gives instructions to the model and then to the view.

For example, for a request to GET '/contacts':

First, the controller asks the model to fetch the data:

```
@contacts = Contact.all
```

The model translates this request into SQL and fetches all the contacts from the database.

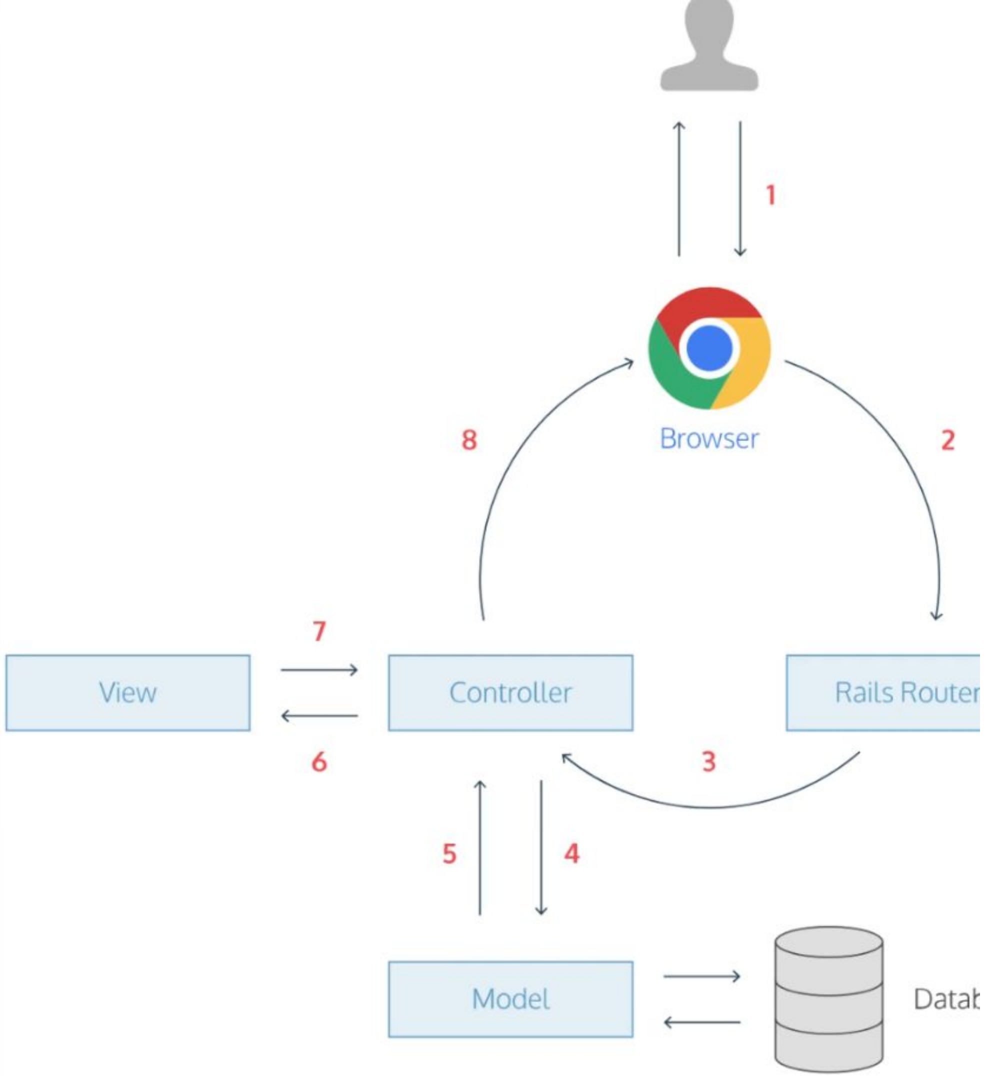


Once the controller has the data, it sends it to the view and asks it to render

For example, for a request to GET '/contacts':

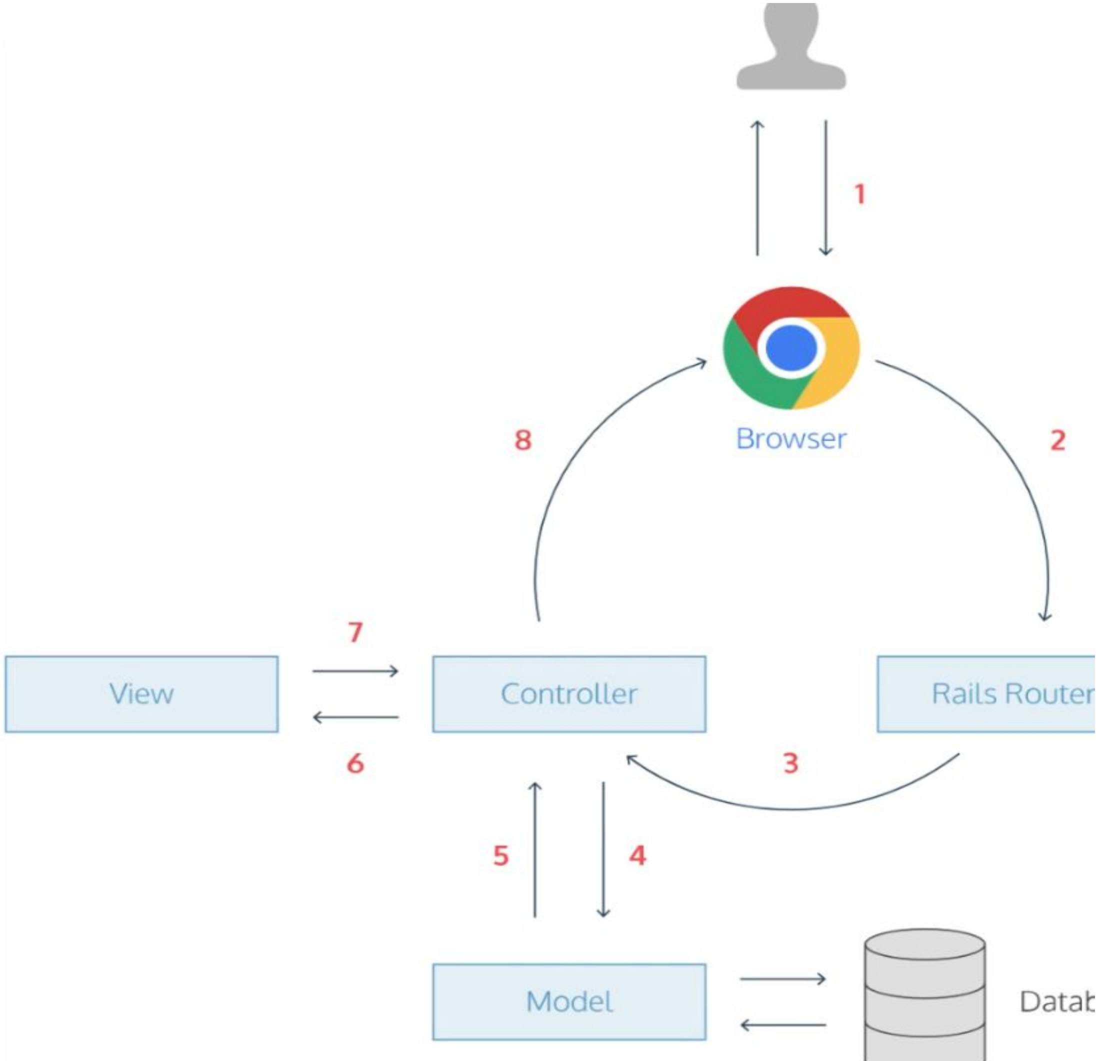
After the model returns the contacts, the controller now has an `@contacts` variable.

The controller shares this variable with the view, and the view generates the dynamic HTML.



This flow never changes

1. User types in a URL
2. The browser sends the request to the Rails server
3. The router forwards the request to the appropriate controller
4. The controller asks the model to fetch data. Sometimes also to create, update or destroy data.
5. The model returns the data
6. The controller forwards the relevant variables to the view
7. The view returns the dynamic HTML / CSV / PDF etc.
8. The controller sends it to the browser



M vs. V vs. C

Model	Handles data and business logic
View	Presents data to the user in the requested format
Controller	Receives user requests and asks the appropriate models and views to carry them out

When in doubt: fat model, skinny controller

Controllers should do as little as possible:

1. Call the relevant model methods
2. Render the relevant view

Logic goes in the model, not the controller

Rails is a gem

Let's install it!

```
gem install rails
```

To save time: `gem install rails --no-ri --no-rdoc`

Rails is a command

If you type `rails` into your terminal, you’ll get instructions on how to use it

Let’s generate a new Rails app:

```
rails new crm
```