What the duck is MVC?

A brief introduction by Kiefer Weis

github.com/kiefer-dev
linkedin.com/in/kiefer-weis

Photo by Ann H from www.pexels.com

Picture this if you will.

You're building a web application to *quack* (track) your beautiful, extensive, world-renowned rubber ducky collection. You threw together your project in a weekend because you thought you could bang this one out - you figured "hey, I can just separate the JavaScript into two files - the client-side and the server-side. That's enough organization, right?" Big mistake, bucko, because now you're lost and scared in a duck pond full of spaghetti code.

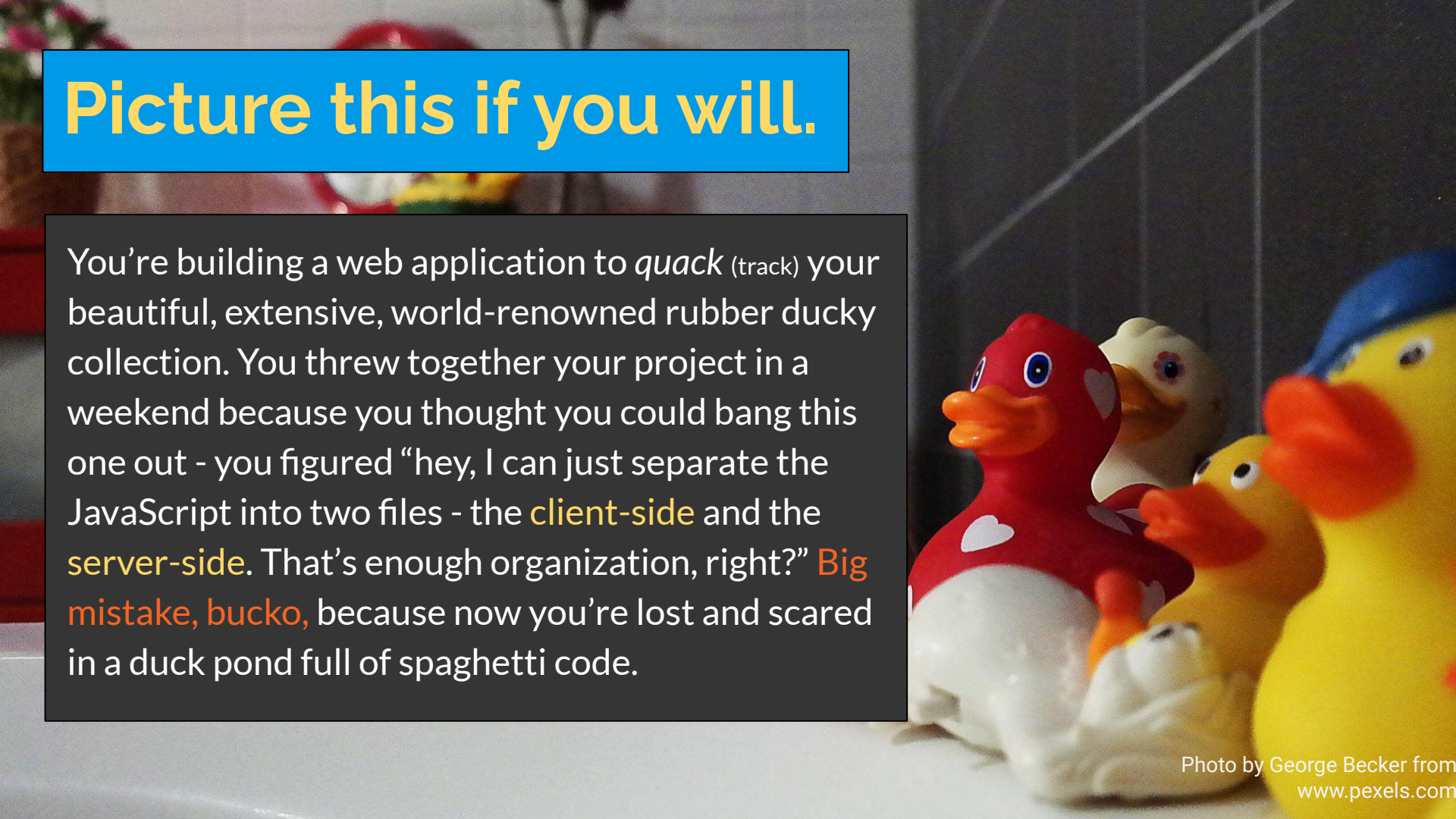


Photo by George Becker from www.pexels.com

—

You didn't want to bother thinking about how you could efficiently structure your program, now it's 11pm the night before the **National Rubber Ducky Convention** and your website is **BROKEN**.

"Where did I go wrong?!" you ask yourself. **"How could I possibly have avoided all this spaghetti??"**

- Should've looked into the **MVC architectural paradigm** before building your program.

You ducked up.

Photo by icon0.com from www.pexels.com

MVC stands for MODEL / VIEW / CONTROLLER. It's simply a way of separating out pieces of your program so that they're more manageable, readable, and modular.

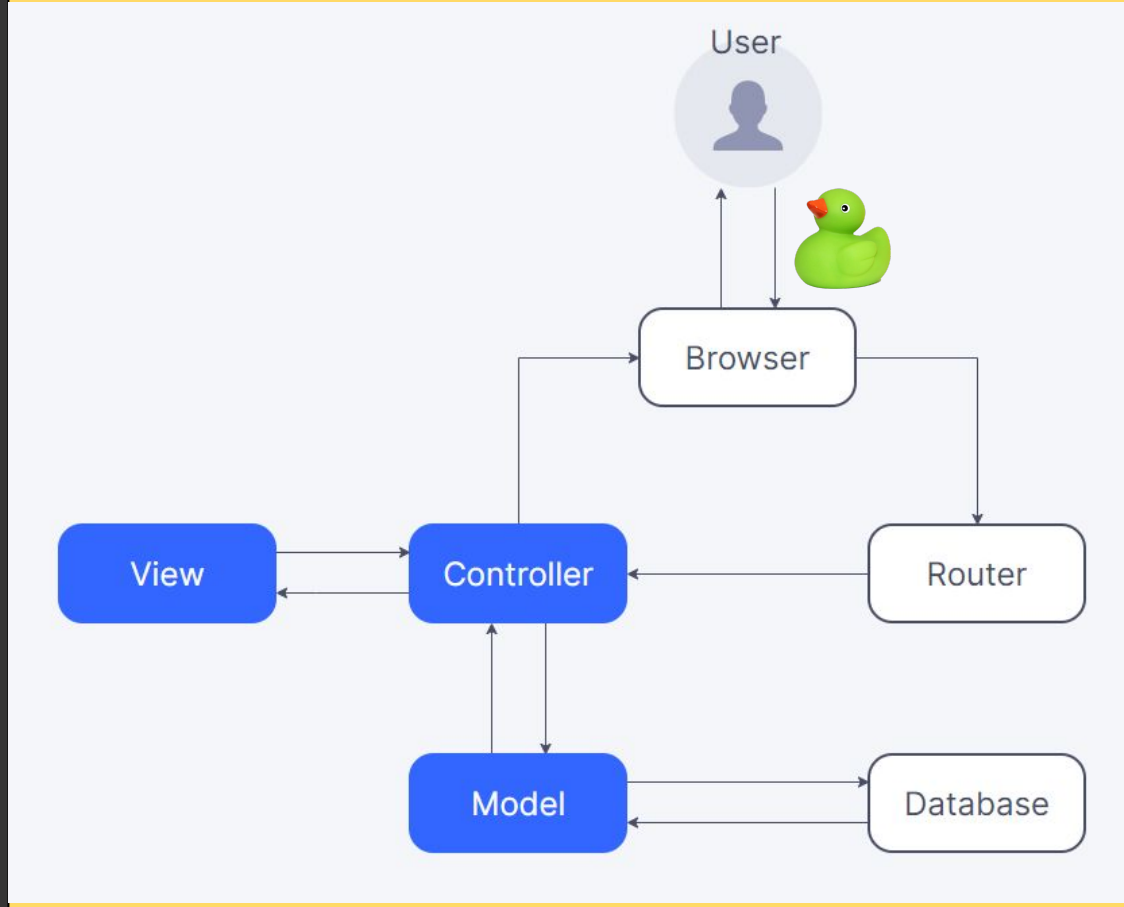
It doesn't require learning anything new - it's just an "architectural paradigm" to organize things!

Even this baby could understand the principles of MVC, if they wanted to!

Photo by Arina Shvets from www.pexels.com

It all starts with the user. Our web application shows a page in the browser that lets the user view every rubber ducky in their personal Rubber Ducky Collection database.

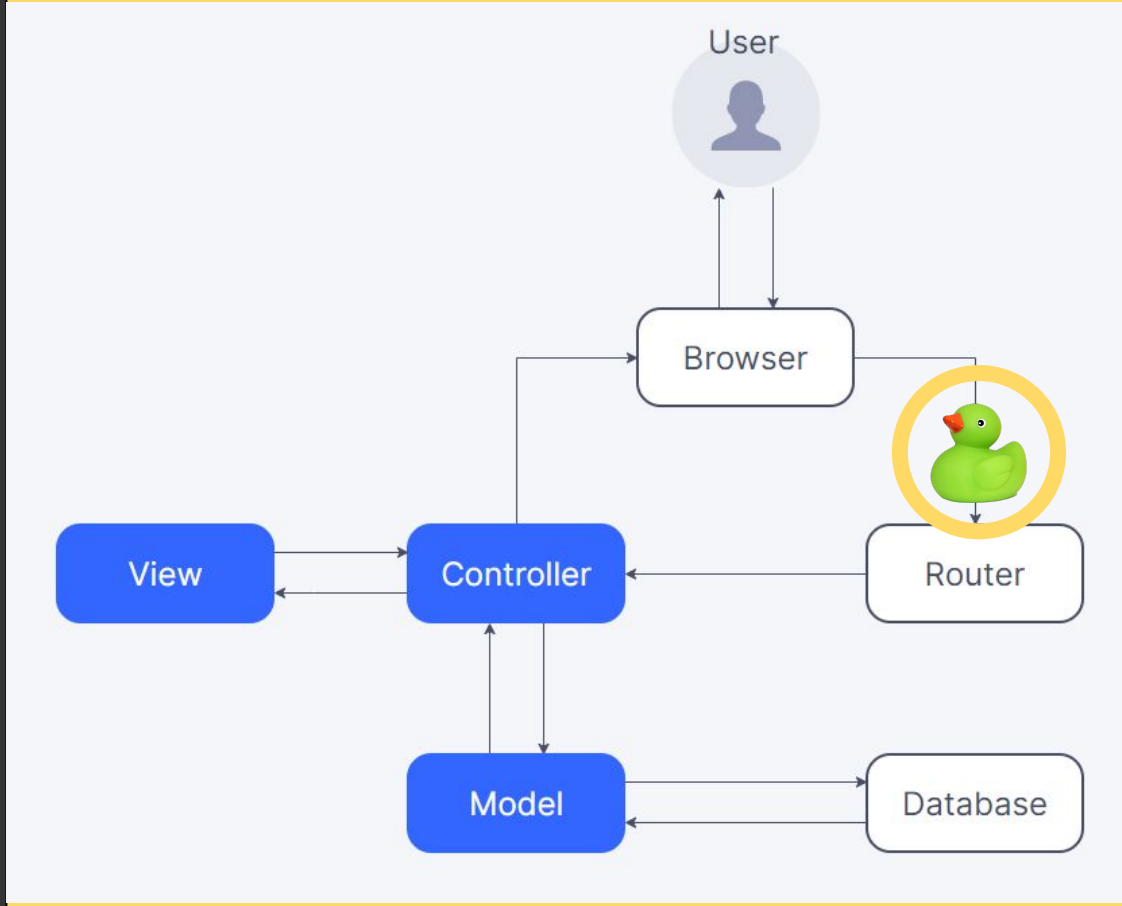
For this example, the user wants to add a brand new rubber ducky to the database.



Flowchart made with zenflowchart.com
Duck image by Anthony from www.pexels.com

The user uses their browser to enter the info about their new ducky (represented by that lovely green duck over there), and clicks “add.” This sends a request to our Router, which is a file we’ve specifically setup to listen for requests from the browser.

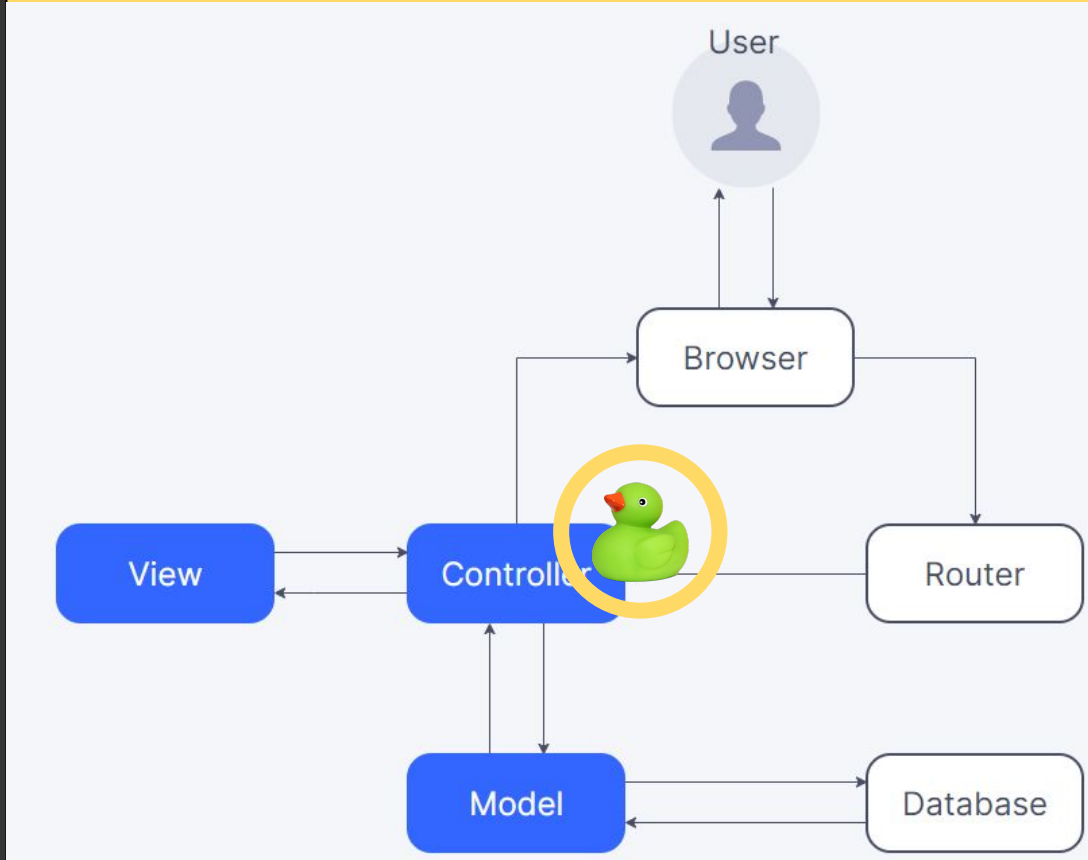
Our request here is represented by a nice yellow circle.



Flowchart made with zenflowchart.com
Duck image by Anthony from www.pexels.com

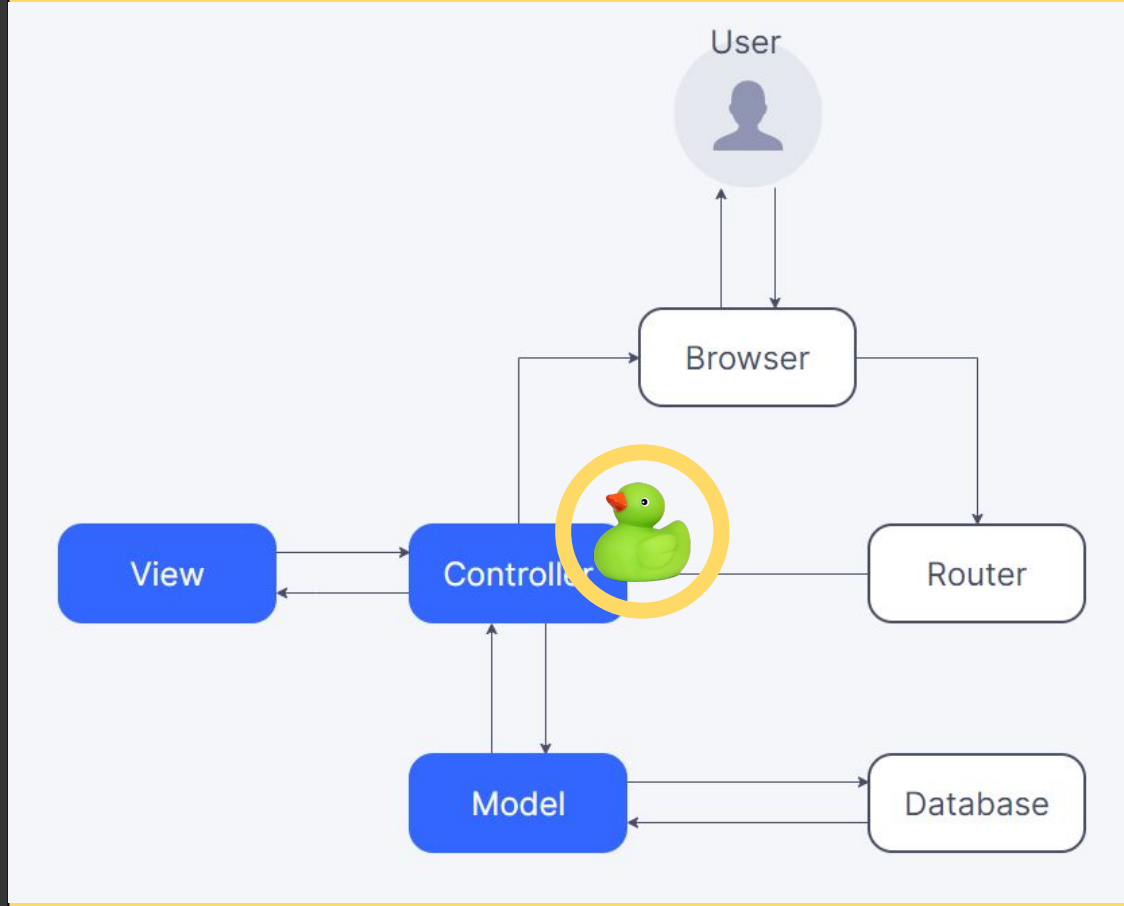
The Router hears the request - it's a request to *add* a ducky to the db, so it's a **POST** request. It also knows that its intended route is to **/ducky.**

All of the logic to hear the request and send it out is still contained within one file, our Router file.
It sends the request along to the appropriate file (a **Controller**).



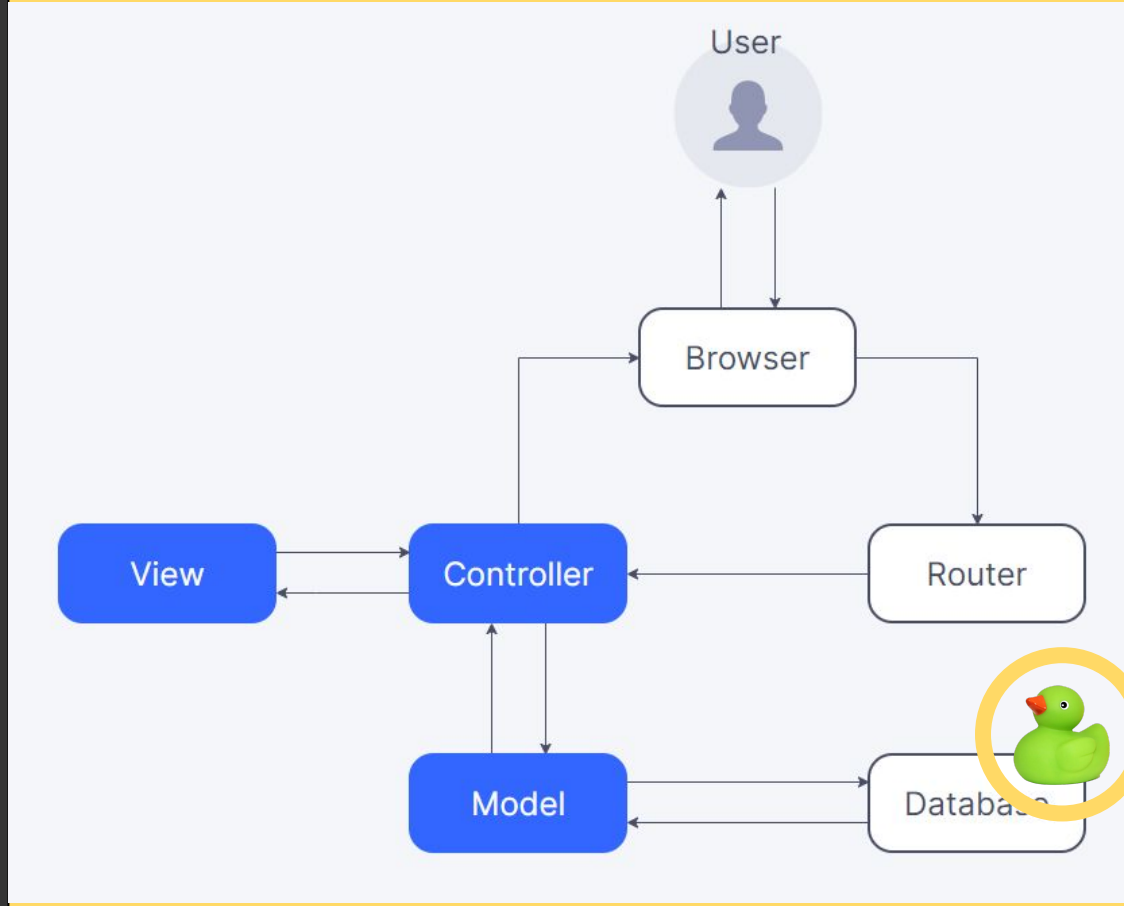
Flowchart made with zenflowchart.com
Duck image by Anthony from www.pexels.com

We have **Controller** files setup for each of our routes - here, the **/ducky** route. This **Controller** file contains functions to handle every possible action on the **/ducky** route, and since our Router knew it was a POST request, we've sent it to the **addDucky** function in our **Controller** file. This function has access to all the data about the new ducky added by our user.



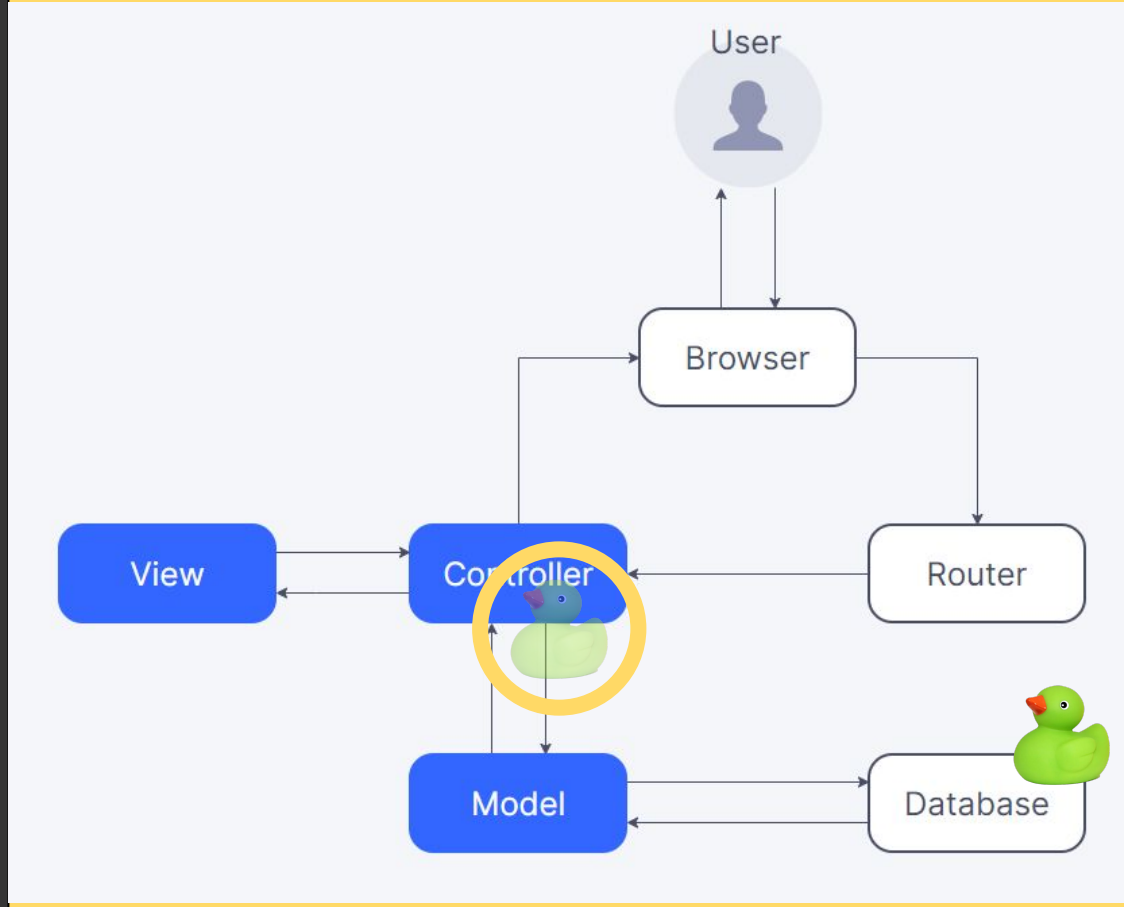
Flowchart made with zenflowchart.com
Duck image by Anthony from www.pexels.com

Our **Model** file includes a *schema* that tells our program how new items should look in the database. The **Controller** file checks with our **Model** file to see how the data should be setup, and passes along the data about the new ducky to the database.
The ducky has been added!



Flowchart made with zenflowchart.com
Duck image by Anthony from www.pexels.com

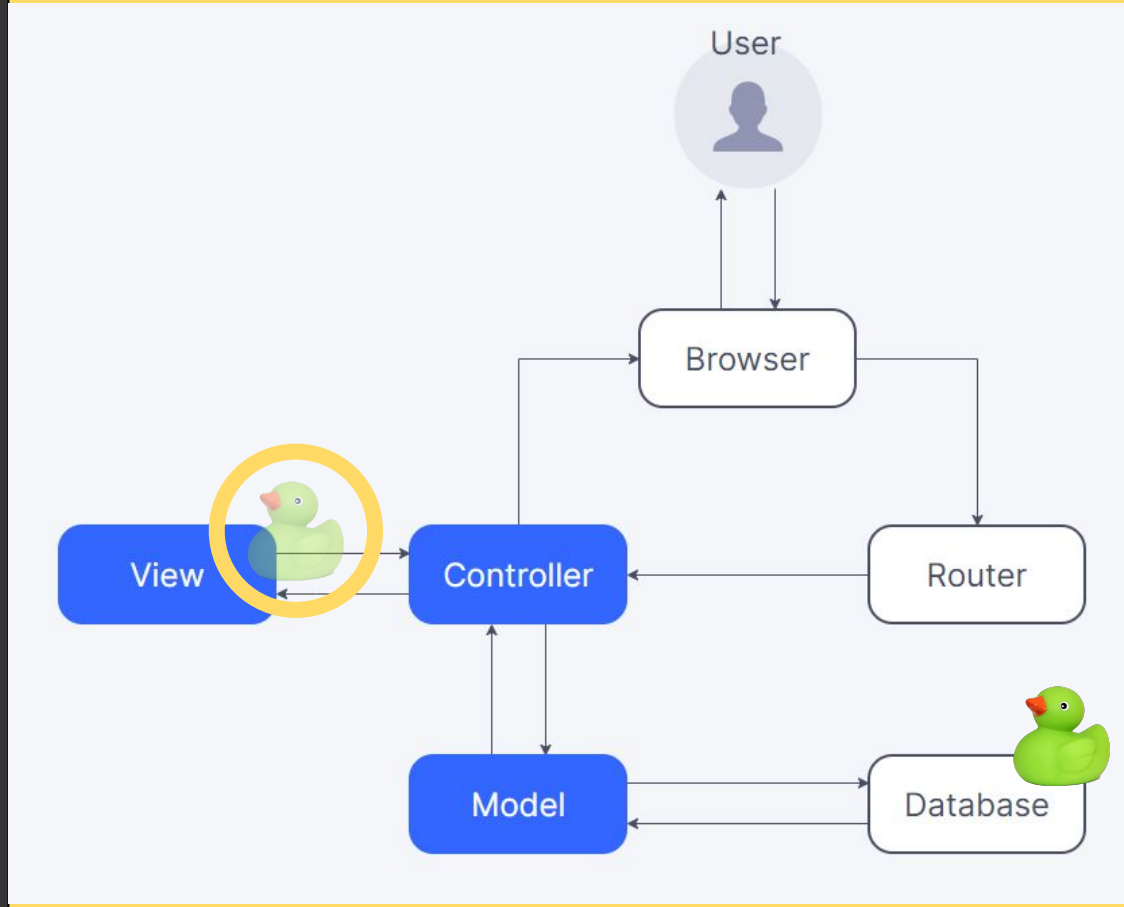
Now the database responds to the **Model**, which lets our **Controller** know that a new ducky has been added to the database.



Flowchart made with zenflowchart.com
Duck image by Anthony from www.pexels.com

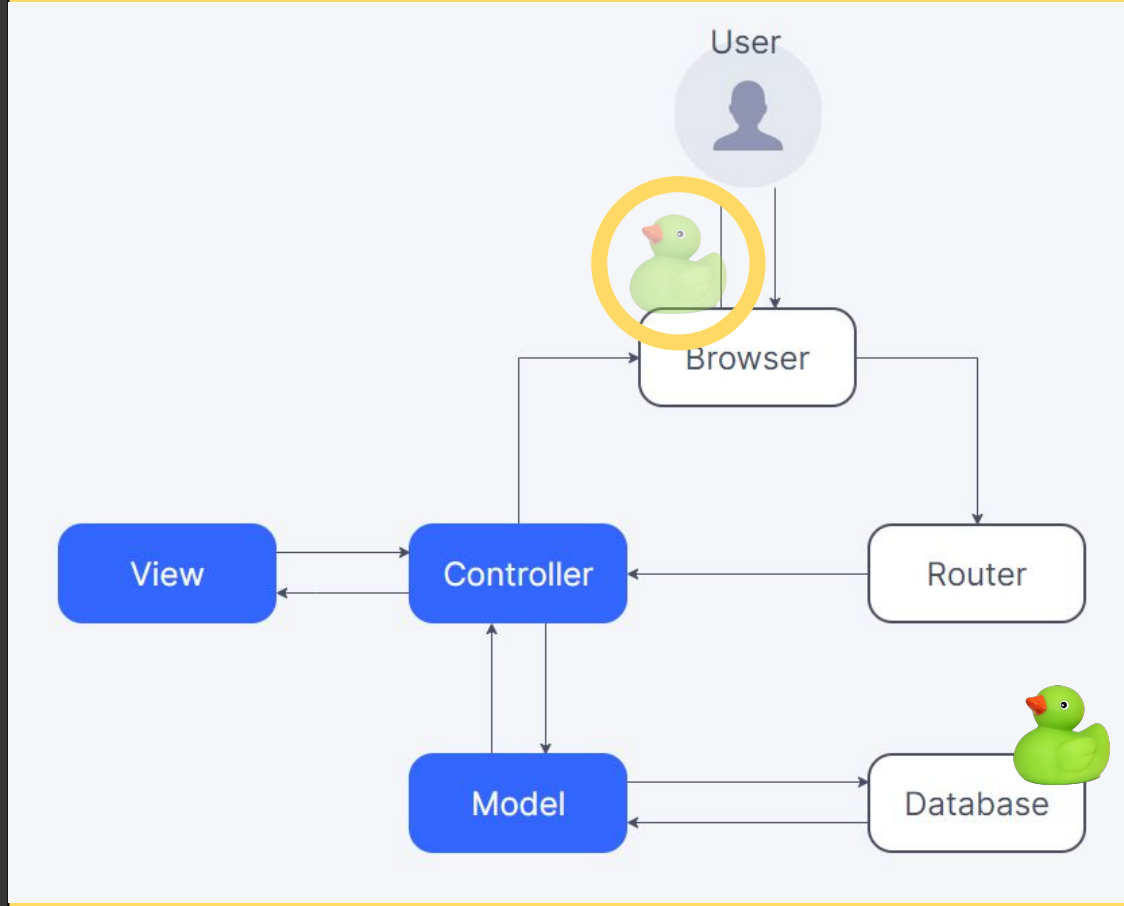
The user needs to know that the ducky has successfully been added, which means their browser needs to update. This is where our **View** comes in.

The **Controller** lets our **View** know that things have been changed, and the **View Engine** updates the "view" by generating new HTML/CSS, using something like EJS.



Flowchart made with zenflowchart.com
Duck image by Anthony from www.pexels.com

The **View** sends this freshly-generated page back to the **/home Controller** (as opposed to the previous **/ducky** controller), which causes the user's browser to update so that they can see it.
Boom! The user is served a fresh page in their browser that shows the new ducky in the database!



Flowchart made with zenflowchart.com
Duck image by Anthony from www.pexels.com

The important thing to note is that each step of this process was separated out into being handled by individual files: the Controllers, the Model, and the Views. We also separated out the logic for directing requests into a specific Router file.

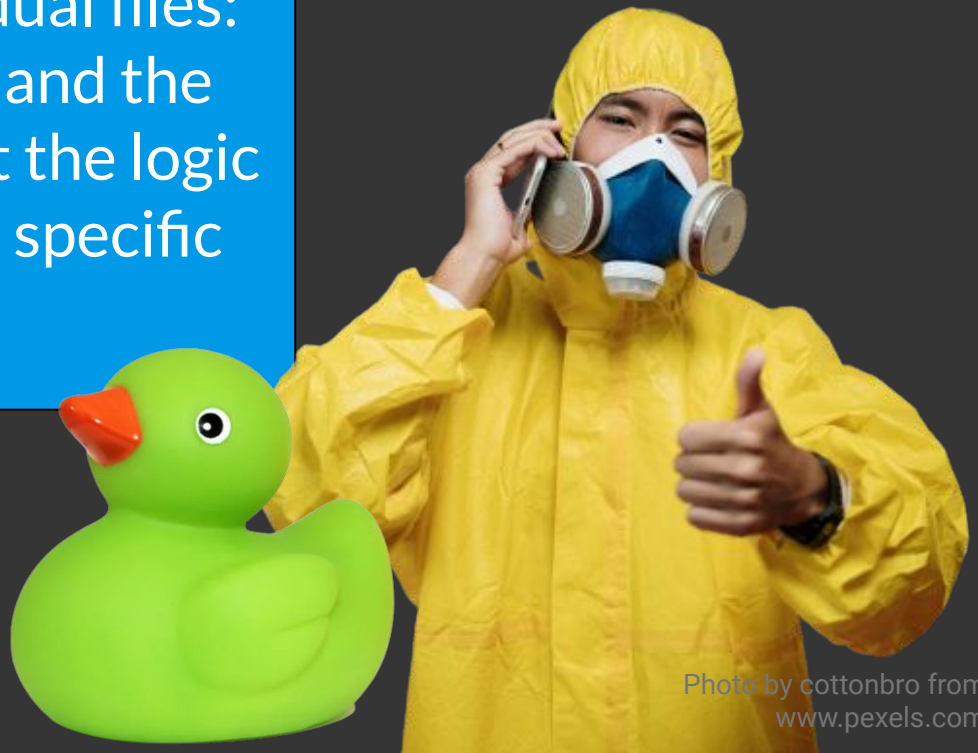


Photo by cottonbro from www.pexels.com

By separating each step of this process into individual M/V/C files, we've organized our program and have saved ourselves a huge future headache.

How, you ask? Well…

Decide you want to collaborate on your project and invite a coworker to look at your program?
No problemo!

Uh oh, a bug occurred at a specific point in the process of adding a new duck to the database? It ain't no thang! You know right where to go to debug!

Photo by Pixabay from www.pexels.com

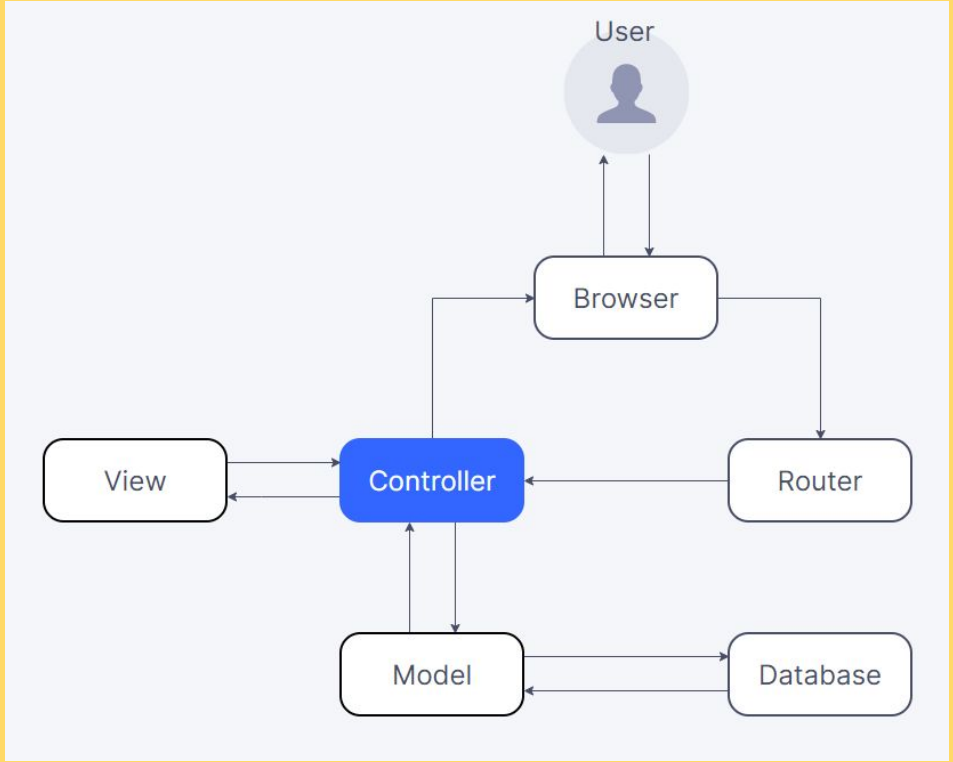
Decide you suddenly hate EJS and also you want to try swapping out your Model? No worries, MVC's inherent modularity means you don't have to worry about messing up the rest of your program!

Photo by Ketut Subiyanto from www.pexels.com

To recap:

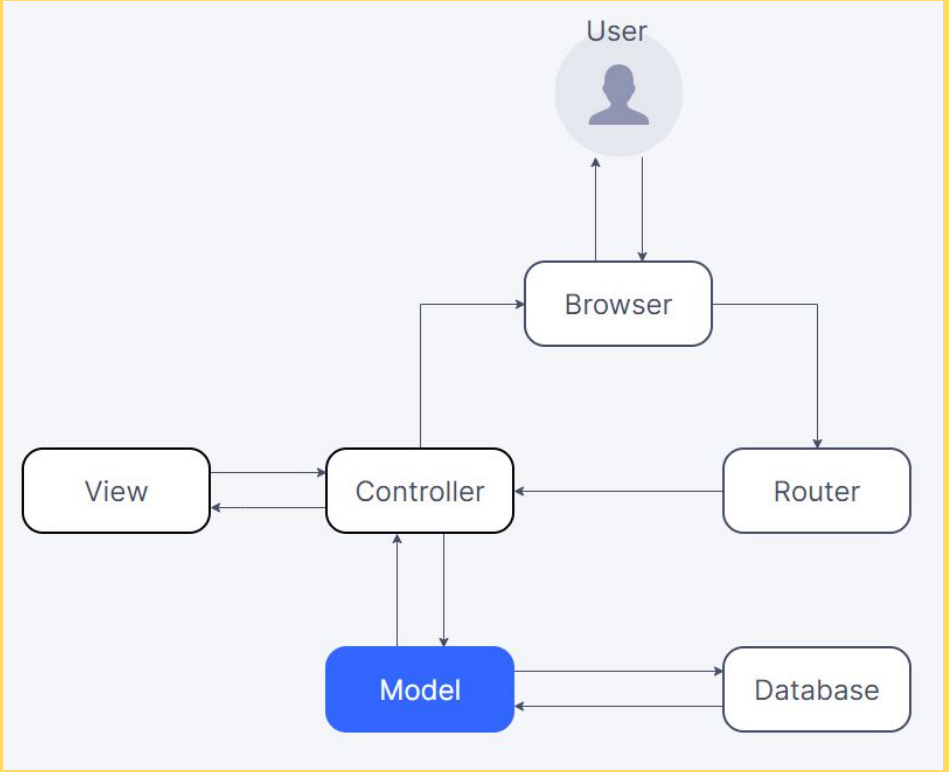
The **CONTROLLER** files hold all of the CRUD logic. The router sends along the user's request from the browser to the proper controller file, based on the request's *route* and *method* (post/get/put/delete).

The **CONTROLLER** talks to both the MODEL and the VIEW.



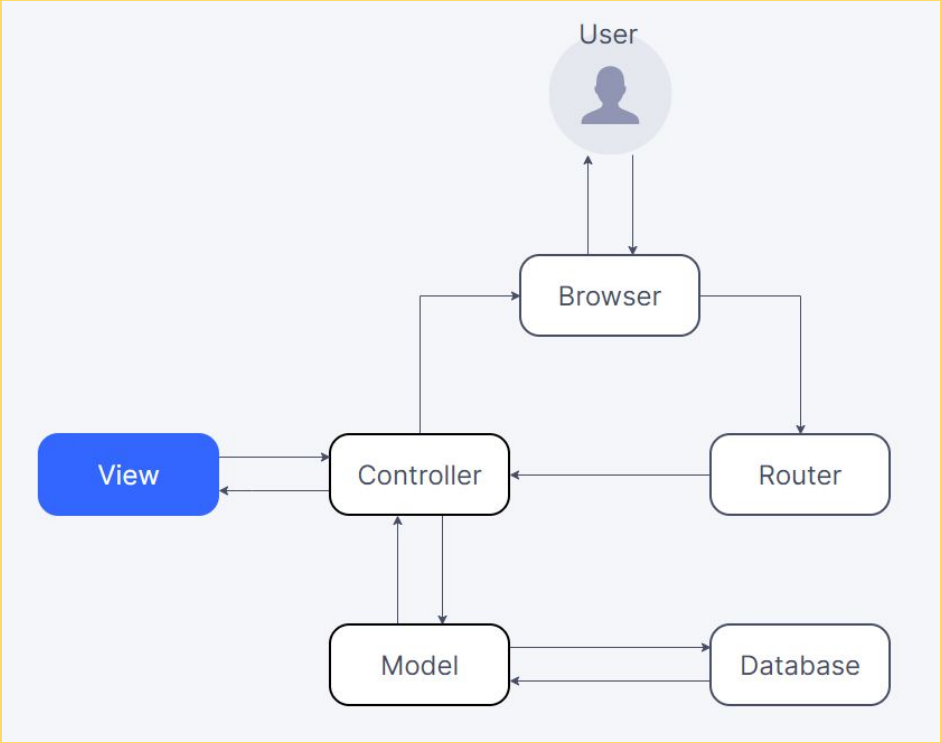
To recap:

The **MODEL** file includes the *model* for how data should look, and sends/receives data to & from the database. It passes this data back to the CONTROLLER.



To recap:

The **VIEW** is what generates the stuff that eventually gets sent to the user's browser for them to see (frontend HTML/CSS). It takes instructions from the CONTROLLER, builds the view, and sends it back to the CONTROLLER to be show to the user.



Web applications should be separated into individual files for the MODEL, the VIEW, and the CONTROLLER!

Otherwise, next time YOU might be the one finding yourself trying to debug a huge trough of spaghetti to fix your program the night before the National Rubber Ducky Convention 😢