First, a review. From Wikipedia:

*Model–view–controller (MVC) is a software architectural pattern for implementing user interfaces on computers. It divides a given application into three interconnected parts. This is done to separate internal representations of information from the ways information is presented to, and accepted from, the user. The MVC design pattern decouples these major components allowing for efficient code reuse and parallel development.*

*Traditionally used for desktop graphical user interfaces (GUIs), this architecture has become popular for designing web applications and even mobile, desktop and other clients.[3] Popular programming languages like Java, C#, Ruby, PHP and others have popular MVC frameworks that are currently being used in web application development straight out of the box.*

*The model is the central component of the pattern. It expresses the application's behavior in terms of the problem domain, independent of the user interface. It directly manages the data, logic and rules of the application.*

*A view can be any output representation of information, such as a chart or a diagram. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants.*

*The third part, the controller, accepts input and converts it to commands for the model or view.*

I think we can assume that all active web developers have a good handle on the MVC paradigm. We understand that the model maintains the current working state of the application, while the view interacts with the user, fetching data from the model, accepting input from the user, and then pushing changes back to the model. The model and the view do not interact directly, they always communicate via the controller.

So models and views are fairly well understood, but the job of the controller is much muddier. In essence, the controller is responsible for ‘all the other stuff’ that does not fit within the domain of the model or the view.

In a Ruby-on-Rails system the controller functionality is provided by ‘controllers’, often mapped to individual models, and ‘routes’, definitions that allow the logic of the controller to accept incoming requests, and route back generated responses. Often, controllers implement request-respond interaction models, where one party (user, using the view) makes all the requests, and the model does all the responding. In this interaction, the model pattern never has the ability to initiate any action whatsoever, it is entirely passive.

This has been the dominant paradigm for web-site development and at least a decade, and has been very successful.

But, there has always been an issue with keeping the view updated with current information, and we all know where the refresh button is on the browser. We have been well trained.

It became clear that browsers needed more programmable power, and the Javascript language has become the dominant language in the browser.

Javascript has its good and bad features, we don’t need to repeat all of them here. It is fair to say that while the JS runtime environment delivered by modern browsers is often a top-notch implementation, the language itself has a set of odd features and off-putting behavior at the best of times, and has some major pitfalls which have burned many a developer.

Let us state here that Javascript is an excellent assembly language for a browser, and that is good. But, when did you last write a complete application completely in assembly language? I have been writing software for 53 years, and my last assembly project was in the late 1970’s. It is true that Javascript is a good language to target, but there are better primary development languages than Javascript.

Ruby is a great programming language, and, via the Opal compiler, can be compiled into Javascript for execution within a browser.

So, we have uniformity of development language across the Model and View environments in ruby-hyperloop, but what about the Controller?

As mentioned above, the standard web development interaction model is request-respond, otherwise known as client-server. But, that model is very constricting when you have basically a peer-to-peer working relationship between the Model on the server, and the View within the browser. Either party should be able to initiate an exchange of information.

Ruby-hyperloop addresses this issue straight on by implementing a bi-directional flow of information between the two cooperating computing environments. **Ruby-hyperloop is a complete re-think of what the MVC Controller should be and how it should be implemented**.

<<More to do...>

Talk about React component model, reactive-memory model, and how the Hyperloop “controller” uses these to tie the attributes of a Model directly into the contents of components displayed by the browser.

<<Much more to do here.>>