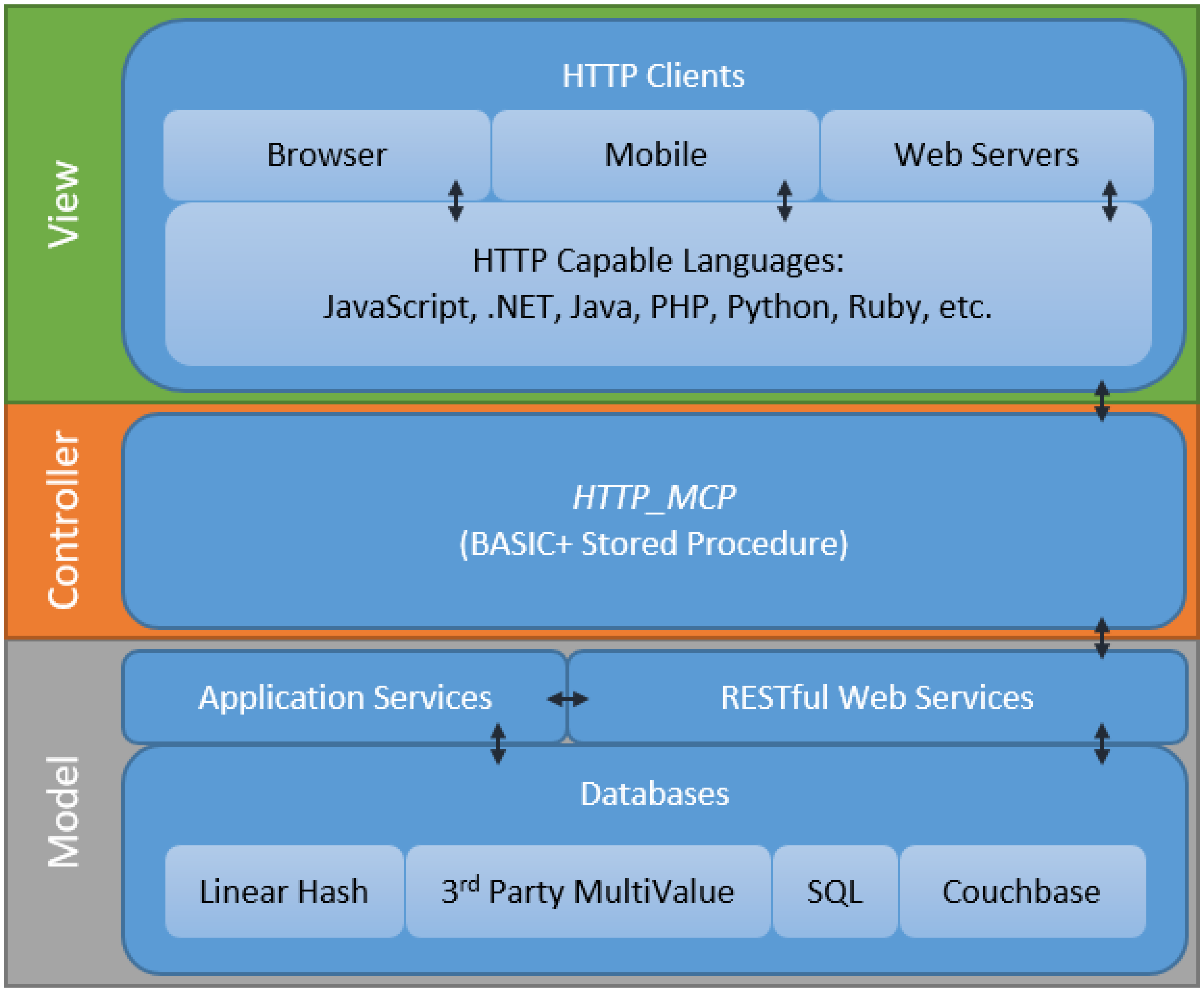
Model-View-Controller

Software systems have been built around the Model-View-Controller (MVC) paradigm since the early 1980s. Much of the adoption of this architecture came about through the growth of web applications. There are many variations of MVC, such as MVP and MVVC, but they all attempt to address a common problem: developing software that keeps important layers separated enough so that changes in one layer will have minimal, if no, impact upon another.

For example, the predecessor of MVC is Client-Server. As you can see, this introduces two layers: a *Client* and a *Server*. This offers some advantages in that the server (where the business logic and database are typically stored) lives independent of the client. However, the client can't be wholly independent of the server. Why? Because the client has to be written to talk to that server's API or data layer. If the server were replaced with a different technology stack or database, then the client would have to be rewritten as well. This "tight-coupling" approach becomes a major thorn whenever any upgrading, scaling, or extension is desired. Every Advanced Revelation developer has felt this pain when attempting to migrate to OpenInsight, and virtually every OpenInsight developer has felt this pain when attempting to bridge their applications into the web and mobile space.

MVC seeks to solve most of this dilemma by introducing another layer: the **Controller**. This layer operates between the **View** (think *Client*) and the **Model** (think *Server*). The controller helps to abstract the other layers, thus enabling more flexibility and better development standards. While some have argued that the basic functionality of the internet is classic client-server, there are others who point out that the use of the HTTP protocol itself performs the function of a *controller*. That is to say, HTTP is the abstraction layer. This is why browsers do not have to be re-written if a web server gets replaced. As long as the URLs remain the same, the *view* (i.e., web browser) and the *model* (i.e., the web server) maintain a healthy separation.

For our purposes, the role of the controller can be broadened to include the OECGI and the pre-configured dispatch routine. By default, this dispatch routine is `RUN_OECGI_REQUEST`, but replaced with HTTP_MCP for the SRP HTTP Framework. These routines are, for the most part, pass-throughs (or brokers) for the incoming HTTP request and then again for the outgoing HTTP response. Thus, the OECGI and its dispatcher are another layer between the view and module. With that in mind, we offer the following diagram to help with the big picture understanding of the architectural design that the SRP HTTP Framework follows:


View
HTTP Clients
Browser
Mobile
Web Servers
HTTP Capable Languages:
JavaScript, .NET, Java, PHP, Python, Ruby, etc.
Controller
HTTP_MCP
(BASIC+ Stored Procedure)
Model
Application Services
RESTful Web Services
Databases
Linear Hash
3rd Party MultiValue
SQL
Couchbase