Rails and Sinatra
An Architectural Comparison

Christopher Wood
Patrick McAfee
Derek Erdmann
Eric Caron
Samantha Shandrow
Yi Jiang

Contents

Problem Domain

Rails and Sinatra emerged from the need to easily develop and deploy web applications written in Ruby. The common, overarching problem they strive to solve is to provide an intuitive way for web applications to handle client HTTP requests by invoking some Ruby code and generating HTML content as a response. They both provide different means for encapsulating the business logic and data layers of an application, but utilize similar web interface layers that are exposed to the client. Internally, both architectures also employ different strategies for handling client HTTP requests, rendering client views, interfacing with databases, and representing the data model for the specific application.

In addition, although they both solve one common problem, they were designed for fundamentally different reasons (i.e. adding a HTTP web interface to an existing application in Sinatra or building a new application on top of a well-defined web interface in Rails). At a high level, Sinatra is a lightweight domain-specific language designed to build an application that dispatches off of a simple set of web URLs. Conversely, Rails is an entire web application framework that encapsulates the dispatch functionality provided by Sinatra, but also provides automated view generation, application model encapsulation, and database integration (to name a few key features). Collectively, these differences make each architecture unique in many regards, and we provide a discussion of these unique properties in the following section.

Frameworks

We now present an overview of the Ruby on Rails and Sinatra architectures, emphasizing their unique functional and quality attributes that make them ideal solutions for specific applications.

Ruby on Rails

System Background and Context

Ruby on Rails is an open-source web application framework. It emphasizes “developer happiness” and favors convention over configuration to promote rapid application development [1]. The ultimate goal was to provide a framework for easy and intuitive application development that provides the “best” possible way to handle common tasks through application scaffolding, router configuration, and database configuration. Rails uses a form of the Model-View-Controller (MVC) pattern to support application development. The core model, view, and controller components of an individual application’s architecture help developers separate business logic from the user interface and promote code reuse and maintainability.

The first version of Ruby on Rails was developed by David Heinemeier Hansson in 2003 while working on the Basecamp project 37signals [2]. It was largely motivated by the lack of web application frameworks that facilitated automated code generation and configuration features. Given the growing popularity of Ruby as a server-side object-oriented language solution, Rails emerged as the common ground to address this problem. The first release was made public in 2005 (Rails 1.0), with minor releases over the next few years that expanded support for other web servers.

Versions 2.0 to 2.3 were released during 2008 and 2009, and added support for more complex view rendering methods, including Ruby formatting for HTML and XML files. Version 2.3 saw the addition of view templates and Rack, a web server interface, which has allowed for a much more diverse selection of web servers and deployment strategies for Rails applications.

Rails 3.0 was released in late 2010 and promotes a greater separation of application business logic and presentation layers. It also introduced the Asset Pipeline, which supports additional processing of static assets such as scripts, stylesheets, and images.

Ruby on Rails has grown beyond its original home at 37signals and has powered many popular web sites and applications, including Twitter, GitHub, Groupon, and Yellow Pages. The current project includes not only the core team and these organizations as its stakeholders, but also developers worldwide who use Rails when developing their applications.

Functional and Quality Drivers

As previously mentioned, Ruby on Rails is driven by "developer happiness" and "convention over configuration." In the context of web applications, these goals translate into the following quality drivers.

- Modular architecture and structure for web applications to enable isolated changes and modular development of web applications.
- Modifiability for the developer to allow them to configure applications after they have been constructed.
- Rapid development speed (i.e. rapid prototyping of web applications).
- Usability of the entire framework at configuration, development, and deployment time.
- Testability for web applications built on top of the framework.

In parallel with these quality drivers, the Rails architecture was motivated by the following set of functional requirements over the years:

- Collectively, the architecture should encompass a full application stack available to programmers based on modularized MVC architectural pattern.
- The architecture should include an automated framework generation process for developers (following conventional standards).

- The architecture should encapsulate database interaction (including ORM services) and RESTful services inside concrete modules (ActiveRecord and ActiveResource).
- The architecture should provide intuitive HTTP request routing and security integration (with an emphasis on RESTful declarations).
- The web server integration should be hidden from the developer, and if possible, rely on existing technologies (Rack has since assumed this role) to hide server logic and improve application portability.
- The architecture should support unobtrusive client-side markup (JavaScript drivers for jQuery, Prototype, node.js) for rich application development.
- The architecture should offer built-in email support to provide easy client communication.

Architectural Solution

The Rails architecture is motivated by a modularized version of the Model View Controller (MVC) architectural pattern, as shown in the data flow view in Figure 1. The model component of the architecture (consisting of the ActiveModel and ActiveRecord) is responsible for the application logic. Its main role is to generate content for a web application view by processing information retrieving from the controller and from the database. The controller component of the architecture (consisting of the ActionDispatch, Filters, and the ActionController) are responsible for handling client requests in order to serve rendered content. Client requests are submitted using the HTTP protocol and are routed to the appropriate action controller (which invokes an action in the model) based on the application routing settings. Thus, we see that the controllers are RESTful, making them less of a bottleneck than the SOAP counterpart. Finally, the view component of the architecture (consisting of the ActionView) is responsible for generating HTML content using embedded Ruby. That is, data generated in the model is passed to the view as Ruby data, which is used to dynamically generate the content of an HTML page, which is then forwarded to the controller to ultimately be sent to the user [1].

Altogether, as shown in Figure 1, client requests are generated by the browser (or another source) and routed through the controller component to the appropriate action controller, which then invokes the corresponding application logic within the model to generate some data, forwards this data to the action view to be rendered, and then relays this new content to the client's browser. Thus, the system is event-driven based on client requests, where the event handlers are encapsulated by ActionController objects that are responsible for directing the behavior of the model and view components of the application.

Perhaps the most significant implication of this MVC pattern is that web applications are built within the context of the Rails framework, rather than querying it for the provided functionality. In other words, Rails provides a boilerplate template for web applications that handles much of the development and runtime issues (such as routing and database ORM). The isolation between each of the separate components and restricted communication paths within the framework also enables increased testability (both in terms of the code and with test-driven development processes), different development

teams and workflows, and application modularity and interoperability.

With this pattern the application developers can also focus their attention on the model and view parts of their application, rather than spend a significant amount of time with the controller. This rapidly speeds up development because the controller component of an application has typically been the most difficult to configure. Furthermore, it promotes high degrees of code reuse with the provided modules and other Ruby gems for Rails.

Another very important tactic used to achieve the desired modifiability in this architecture is the presence of the ActionDispatch, which encapsulates routing decisions that can be changed at compile-time with minimal effort.

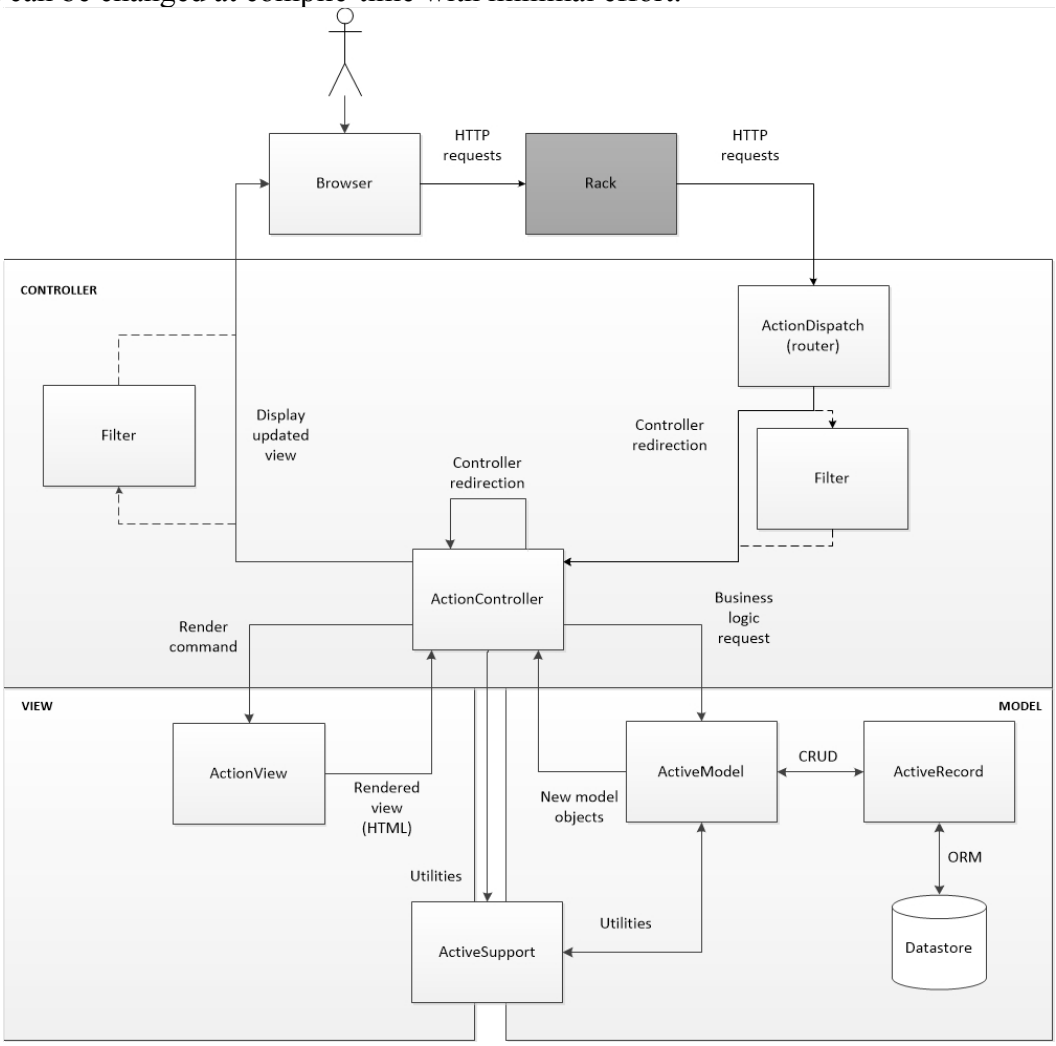

Figure 1: Dataflow view of the Rails architecture.

We can also visualize how Rails supports the MVC architecture in the context of a layered view, as shown in Figure 2. This view highlights the modularity of the Rails architecture, and it also highlights the top-down approach used to build applications within the context of Rails. Many tedious tasks such as database configuration and management, security integration, and testing, are all provided by lower-layer modules.

Application development happens on top of these modules, either within the same layer or at a layer above, in order to reuse the functionality provided by lower layers.

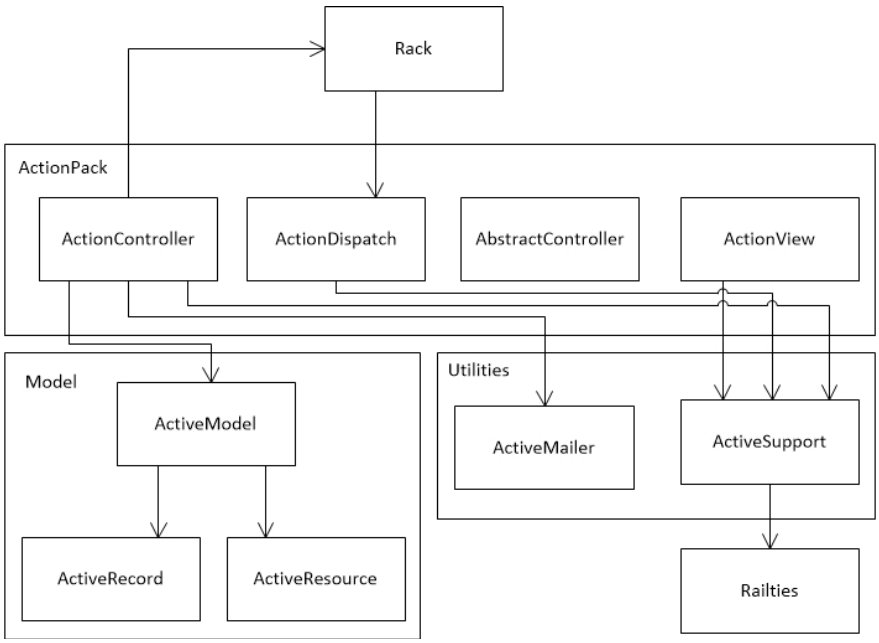

Figure 2: Ruby on Rails layered view, showing the logical dependencies between the various modules that make up the architecture. An arrow indicates directional dependency.

Table 1 shows the responsibilities for the major layers in a standard application deployment.

Table 1: Layer responsibilities in the Rails architecture

Layer	**Responsibilities**
Rack	Intercepting client HTTP requests and forwarding them to the Rails ActiveDispatch layer. Forwarding HTTP requests from the Rails ActionController to the client browser.
ActiveDispatch	Intercepting HTTP requests from the Rack layer and routing them to the appropriate ActionController (i.e. invoking the appropriate handler function).
ActionController	Invoking the appropriate methods in the Model layer to update the state of the application, passing this data along to the ActionView so that it can be rendered (this relationship is omitted from the diagram), and forwarding the HTTP response (as a type of rendered view) to the Rack layer so that it is sent to the client.
ActionView	Building static views based on data provided by the ActionController,

	which ultimately comes from the Model layer.
Model	Encapsulating the application logic, including all interaction with the application databases (via ActiveRecord) and with other web services (via ActiveResource). This is where developers implement their application.
Utilities	Providing standard functionality to Rails application developers, such as email support via the ActiveMailer module.

This view also introduces the Railties component in Rails, which is the backend tool responsible for initializing each Rails process through the command-line interface. The most common usage of the Railties module is scaffolding, which is the same as automatically generating the template MVC code for an application. Thus, from the layered view we see that Rails encompasses everything from the runtime behavior of an application to the development time configuration modules necessary to build such applications.

Assessment

The Ruby on Rails architecture has several notable strengths. It is extremely modular, with well-defined and well-engineered boundaries between different modules, and through its structure is able to promote the same engineering style to application developers playing in the Rails sandbox. These modules allow for the encapsulation of many common development tasks. For example, the venerable ActiveRecords provide commonly abstracted interactions between the coded model and the raw data. It can also manage security, routing, and other such tasks which are ever present concerns that, if attempted regularly, become issues that cause massive amounts of overhead to occur. This extensive modular structure with built-in support for these common web development tasks also allows for good use of the DRY principle (Don’t Repeat Yourself).

This entire structure strongly encourages several key tenets of good design for applications in the framework - notably, the use of Model-View-Controller architecture and similar project source layout. Thanks to the previously mentioned functionality, it also allows developers to (for the most part) ignore the underpinnings of what they are building and can focus on the specific details of their application. It also allows for the quick creation of scaffolding for both new users and veterans, due to the structured convention, which can speed up development.

On the flip side, Rails provides a very large technology stack to work with, which can be intimidating. Breaking out of the prescribed model can be very difficult and time consuming. Lighter applications which need not use all the functionality may find Rails to be a burden. Furthermore, Rails is well known to scale poorly, to the point that Twitter was forced to forego the framework as they expanded. Some of this is undoubtedly connected to the use of the ActionController as a mediator of sorts, it being central

to all the other components. Concurrency and threading support are also subpar, as the architecture itself is single-threaded. All HTTP requests are buffered into a single request queue that are handled sequentially, thus serving as a bottleneck to application throughput and performance.

To compound these issues, porting previously written code into Rails is a daunting and frequently infeasible task. Another issue is that, since the web application is built within the context of the Rails framework, major architectural changes may require massive changes to the application model itself. Rails applications have a tendency to depend on the functionality exposed by critical Rails modules, so any changes to such modules will inevitably cause increased maintenance costs. Thus, scalability (in terms of architectural changes) is rather poor.

Despite these flaws, Rails still does an admirable job of meeting its goals. Its architecture allows for extreme modularity, in both in the framework itself and in applications built for it. Changes to one cohesive part of Rails will minimally affect other sections, if at all. The choice of late binding drives the ability for developers to plug in in their own application into the framework. Systems can be brought online within minutes (short of errors in code, of course). Rails also manages to extends a whole host of services via the API, such as mail services. Despite such issues as poor portability into the framework, Rails does exactly what it meant to do. It was designed with a specific usage in mind, and admirably adheres to this goal.

Sinatra

System Background and Context

Sinatra is an open source web application library and domain-specific language developed by Blake Mizerany in 2007 and has since been evolved by other web developers [3]. Sinatra is written in Ruby and provides a web application micro-framework inspired by (but not similar to) Ruby on Rails, Rango, and Camping.

When Mizerany designed and developed Sinatra, he focused on flexibility and size. Sinatra is very small and emphasizes “quickly creating web-applications in Ruby with minimal effort”. To achieve this goal of minimal size and flexibility, Sinatra wraps a lightweight HTTP request and response layer on top of the Ruby independent web server interface, Rack. It also does not use the typical model-view-controller design pattern like Ruby on Rails and other web application frameworks. Sinatra provides the following benefits for web application developers:

1. Sinatra is not tied to any particular JavaScript framework, MVC paradigm, or templating system, leaving the developer in complete control while developing applications.
2. Development is flexible since Sinatra does not force the developer to use helper functions and generator scripts.
3. There are no complicated setup procedures, configuration, or generators, which allows the developer to quickly start implementing a functional web app.

Sinatra is financially supported Travis CI and is used by companies such as Apple, BBC, LinkedIn, and GitHub. The primary stakeholders for Sinatra are web developers, web users, and Blaze Mizerany.

Functional and Quality Drivers

Sinatra is driven by the end goal of having a simple platform that allows developers the flexibility to create architecture in whatever way they see fit. Sinatra's goals translate into the following quality drivers:

- Flexibility to allow users (application developers) freedom to organize their application architecture as desired.
- Performance to build HTTP web services with high response throughput.
- Simplicity, stability, and maintainability to support isolated changes with small ripple effects on user applications in the event that the the Sinatra architecture should change.
- Supportability of the entire architecture to provide the most up-to-date services.
- Usability of the provided modules.

In parallel with these quality drivers, Sinatra's architecture was motivated by the following set of functional requirements:

- Provide a lightweight module to handle HTTP requests and responses that can be utilized by existing applications.
- Create web applications on top of a Rack server.
- Fill a niche where there were no prominent solutions for at the time for a lightweight Ruby framework that allows developers freedom with their architecture.

Architectural Solution

The dataflow view for Sinatra is shown in Figure 3 below. There are three separate flows shown by the different colored modules and arrows.

The core data flow starts when the browser or another server outside of the Sinatra system makes a HTTP request that goes through Rack, still outside of the system, and then sends the GET/POST request to the application. Next, the application sends a data object response to the browser/server. This flow also includes all redirects that go from the URL to the redirect module and sends a server URL back to rack and thus the browser/another server [3].

The stream dataflow is used for uploading and downloading content. From the application a streamed request/response is sent to the helper module which then sends a parsed request/response to the stream which then sends a data stream back to the application. This occurs in a looping fashion while there is constant communication through FTP with either a browser or another server through Rack. The benefit of the

external stream module is that it enables more explicit control of the data streaming process through a scheduler. That is, the client to the stream module can provide a scheduler that can call or defer data streaming based on the current load of the application.

The last aspects of the Sinatra dataflow include the optional functions provided by the CommonLogger, ExtendedRack, databases and templates. CommonLogger is used for logging activities. ExtendedRack is used when certain middlewares require extended protocols. Templates take the name of a "template" to render and return a string with the rendered output. Database is used for storing files in case of uploads, and accessing files in case of downloads.

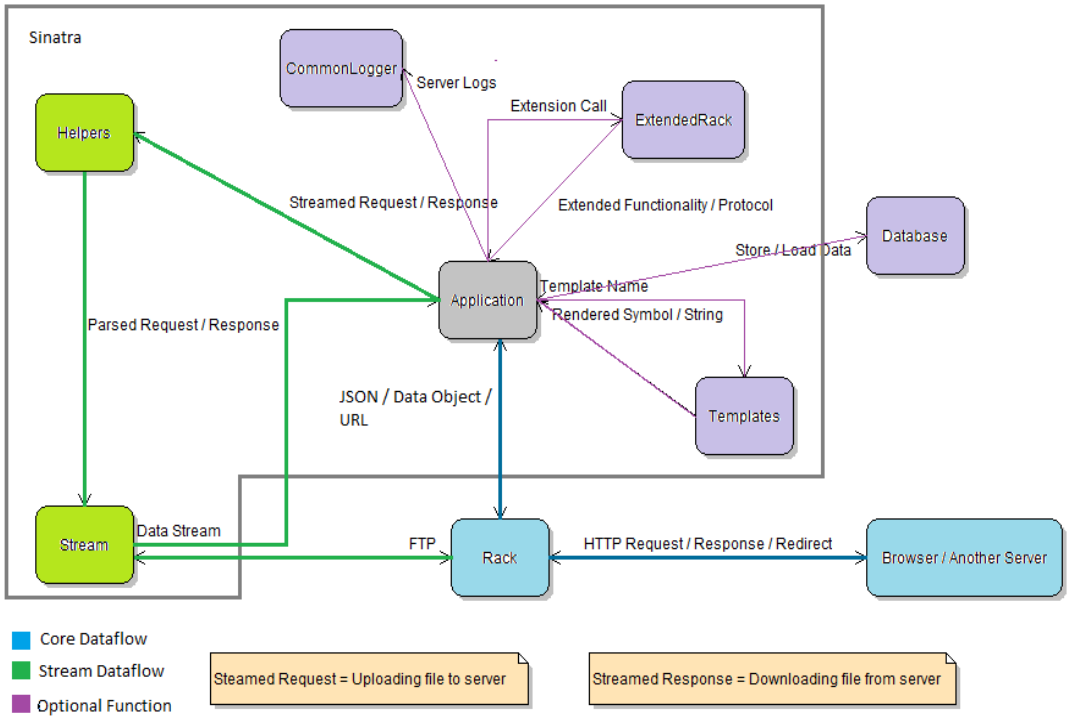

Figure 3. Dataflow view for Sinatra.

Web Requests come into Sinatra through the Rack module. The main Sinatra module includes various components to parse the request and compose the response that is sent out through the Rack. Helpers can be used to parse HTTP headers and or coordinate streaming of data between the browser and the application. Templates are used to render symbols and strings into formats such as SASS. The usage of Helpers and Templates is optional, but the two modules are part of the Sinatra module while Rack is external to the system. It is important to note that the Stream module (as shown in the previous dataflow view of the architecture) is encompassed as part of the Helper module. We chose to not break this view down to any further level because we wanted to capture the same scope as in the layer diagram of the Rails view.

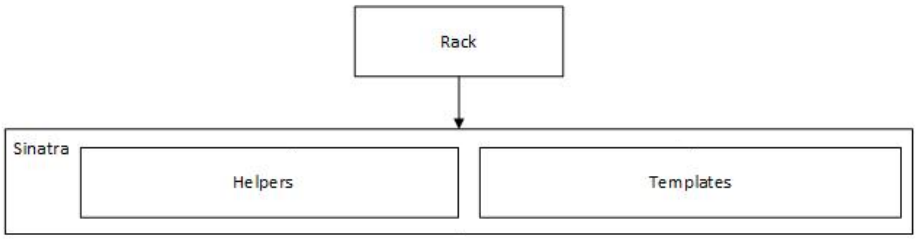

Figure 4. Layer view of Sinatra showing the logical dependencies between the various modules that make up the architecture. An arrow indicates directional dependency.

Assessment

Sinatra's dataflow view highlights the framework's minimalism, as well as its ability to assist the developer with other common tasks. The primary flow (highlighted in blue in Figure 3) includes very few components, allowing developers to use the request handling as a frontend for virtually any application, but other components such as templating engines can be facilitated by the framework.

The layer view illustrates Sinatra's real role as a layer of some other application's architecture. Sinatra is designed to be used for handling requests for other applications, so its existence as a single cohesive unit helps it be used in another application, rather than functioning on its own. The layer view also displays Sinatra's simplicity quality attribute, which is exemplified by the single layered architecture. By having a single layer, applications can easily integrate it into their own existing architecture to provide simple and intuitive HTTP web server functionality.

The dataflow view also highlights Sinatra's flexibility. The flexibility is indicated by the purple modules: Common Logger, Extended Rack, Database, and Templates. These four modules are not required for a web application which adheres to Sinatra's lightweight framework, but they can be added to an application easily for additional functionality. For example, if developers want to incorporate a database into their Sinatra applications, they could include an object-relational-mapper into their application without interfering with Sinatra. This additional functionality flexibility that Sinatra provides benefits developers as they are not forced to use other tools, but are free to use them as needed.

The data flow view also helps indicate, at a high level, the performance qualities Sinatra provides. By requiring so few components for a fully functional application, the scope of Sinatra is rather small (in comparison to other frameworks), aiding in its performance.

A downfall of the Sinatra architecture is that it does not provide a full web development software stack. This is shown in both the layered and data flow views, where some components of the stack are not part of Sinatra. Perhaps the main components missing from the Sinatra architecture include the database and model. With enough time and

effort, however, these missing components can easily be incorporated into an application due to Sinatra's flexibility. Thus, Sinatra provides a simple architecture that requires developers to design and implement only what they need, and nothing more.

Comparative Analysis

Commonalities

Both Rails and Sinatra were designed to help software developers build web applications written in Ruby. Perhaps the largest similarity between these two architectures is the way in which HTTP requests are routed to the appropriate handler and responses are relayed to the client. In both cases, client requests pass through the Rack module before they are sent to the appropriate handler (the ActionController in Ruby and the generic Application in Sinatra). Then, once the handler deals with the request, an appropriate response is sent back to client (which usually consists of a new HTML page to render in the client browser). The routing of requests is also accomplished in in similar ways between the two architectures. Rails requires users to modify a routing configuration file to change how requests are handled (which ultimately have to point to an ActionController method), and Sinatra requires used to just implement handler methods. In both cases, routes are bound at runtime based on the contents of the routing configuration file (Rails) and available controller handler methods (Rails and Sinatra). Furthermore, both architectures allow both static and dynamic (i.e. stream) responses to the client.

Another major similarity is that both architectures handle requests within a single thread, making them not very scalable in terms of managing heavy traffic loads. To our knowledge, there does not exist any built-in support for concurrent request handling (although there are probably Ruby gems available to modify the Rack interface to support this capability).

Aside from how HTTP requests are handled at the web interface level and the single-threaded nature of the architectures, there are no other major commonalities between these two web frameworks. As we have shown in the previous sections, both were designed to solve largely different problems with two very unique solutions.

Rationales for Differences

Behind the layer at which HTTP requests are intercepted, Rails and Sinatra are significantly different. Both of these projects used different architectural patterns and strategies to achieve their unique quality and functional requirements. Rails is built around an MVC architecture that provides physically independent modules that cooperate to handle application logic, client request handling, and user interface rendering. It is a full-fledged framework for the entire web application stack. Conversely, Sinatra is conceptually a language for defining a single event-driven module that retrieves HTTP requests, performs some application logic, and then returns a response to the client. Thus, Sinatra is a micro-framework that embodies HTTP request/response logic for applications to leverage.

This structural difference highlights the key rationales between both the different Rails and Sinatra architectural solutions. Sinatra is strictly meant to be a module to provide event-handling for client HTTP requests, and this module can be used as part of an existing Ruby application. It was designed as a domain-specific language for building web applications, which means that it is to be used by developers to add HTTP web support into an existing Ruby application. Conversely, Rails is meant to encompass the entire web application stack, including both the event-handling operations for client HTTP requests and underlying Ruby application within the model component. In a way, the main distinction between these two purposes is that Sinatra was designed as a means of integrating a web interface with an application, and Rails was designed to build an application on top of a well-defined web interface. Because of this difference, Sinatra does not provide many modules that are of use to the developers, which is the exact opposite case with Rails, where many modules for database management, security, view rendering, etc are provided for the developer.

One implication of this difference is that Rails typically has a steeper learning curve than Sinatra, as a developer must learn the entire application stack (including all model, view, and controller components). In Sinatra, one must simply learn the HTTP client handler API and implement the appropriate request handler functions.

There is also a subtle difference in the architectures that specifies how dynamic data is streamed to the client. Rails integrates this functionality into the ActionController so that it is responsible for both static and dynamic responses, whereas Sinatra offloads this streaming capability to a Stream module. Rails couples the generation of static and dynamic responses in order to maintain adherence to the application MVC architecture. Conversely, Sinatra decouples the generation of these responses for concurrency purposes. The Sinatra

When to use Ruby on Rails

As a framework that supports the development of new applications adhering to the MVC architectural pattern, Rails is most appropriate in the following scenarios:

- A new web application is being launched from no existing code base. Rails encompasses all of the application logic and enforces good coding practices with its architecture from the very beginning. Furthermore, merging existing code with a Rails application is not a straightforward process.
- Automatic database integration, email support, logging, caching, session management, and or test generation are required. Rails stresses the DRY principle, and by providing many auxiliary features the developer is not forced to roll their own implementations or integrate third party libraries to provide this support.
- Prototypes for web applications need to be released for deployment quickly. Rails code generation (scaffolding) provided by the Railties module automates many of the tedious tasks needed to be done by web developers to get an application up and running (e.g. database configuration). By providing tools to favor convention

over configuration, Rails also promotes agile development processes with rapid, incremental releases. Also, deployment is very easy to do with service providers such as Heroku.
- Community support and continual service is important to the application developers. Rails has a very extensive development and support community, and they both contribute to the rapid advancements in the project.

When to use Sinatra

As a domain-specific language that is used to quickly extend existing Ruby applications with HTTP services, Sinatra is most appropriate in the following scenarios:

- When you need flexibility in your system to create your own architecture since Sinatra contains "no preconceptions of how you organize your domain or business logic" [4].
- When the application needs to only respond to a few well-defined HTTP requests that are dispatched by specific URLs. Sinatra enables developers to very easily define dispatch methods for HTTP requests with the inclusion of a single gem proper method signature.
- When developer libraries need to be quickly tested. This is because Sinatra enables all of the test logic to be embedded in single file, as opposed to an entire Rails application.
- For creating applications that are intended to be an API without frills because Sinatra allows for simple interfacing and redirecting through the use of HTTP requests. It is particularly useful in the context of RESTful APIs. Also, since Sinatra has a "add only what you need" attitude, omitting the view component of an application leads to a smaller project code footprint and less complexity to handle. Furthermore, the API can be changed simply by modifying the dispatch methods available to handle HTTP requests.
- The web application must be independent from the HTTP web interface. The small, simple, and lightweight properties of Sinatra isolate any changes to its architecture so that the web application need not change how it works internally.
- For a basic platform that is simple and has few client endpoints (i.e. micro web applications). Since a developer must specify all of the dispatch methods for every entry point, managing these methods becomes an easy task if there are very few endpoints into the application.

References

[1] "Ruby on Rails." 2004. 1 Nov. 2012 <http://rubyonrails.org/>
[2] 37signals, *About 37signals*. [Online]. Available: http://37signals.com/about
[3] "Sinatra." 2007. 1 Nov. 2012 <http://www.sinatrarb.com/>
[4] Ruby Source. *Rails or Sinatra: The Best of Both Worlds?* [Online]. Available: http://rubysource.com/rails-or-sinatra-the-best-of-both-worlds/