information

*Article*

# Improving Ruby on Rails-Based Web Application Performance

**Denys Klochkov and Jan Mulawka ***

Institute of Computer Science, Warsaw University of Technology, Nowowiejska 15/19, 00-665 Warsaw, Poland; denisklochkovx@gmail.com
* Correspondence: jmulawka@ii.pw.edu.pl

**Abstract:** The evolution of web development and web applications has resulted in creation of numerous tools and frameworks that facilitate the development process. Even though those frameworks make web development faster and more efficient, there are certain downsides to using them. A decrease in application performance when using an "off the shelf" framework might be a crucial disadvantage, especially given the vital role web application response time plays in user experience. This contribution focuses on a particular framework—Ruby on Rails. Once the most popular framework, it has now lost its leading position, partially due to slow performance metrics and response times, especially in larger applications. Improving and expanding upon the previous work in this field, an attempt to improve the response time of a specially developed benchmark application is made. This is achieved by performing optimizations that can be roughly divided into two groups. The first group concerns the frontend improvements, which include: adopting the client-side rendering, JavaScript Document Object Model (DOM) manipulation and asynchronous requests. Another group can be described as the backend improvements, which include implementing intelligent, granular caching, disabling redundant modules, as well as profiling and optimizing database requests and reducing database access inefficiencies. Those improvements resulted in overall up to 74% decreased page loading times, with perceived application performance being improved above this mark due to the adoption of a client-side rendering strategy. Using the different metrics of application performance measurements, each of the improvement steps is evaluated with regards to its effect on different aspects of overall performance. In conclusion, this work presents a way to significantly decrease the response time of a particular Ruby on Rails application and simultaneously provide a better user experience. Even though the majority of this process is specific to Rails, similar steps can be taken to improve applications implemented with the use of other similar frameworks. As the result of the work, a groundwork is laid for the development of the tool that could assist the developers in improving their applications as well.

**Keywords:** web application; web development; application performance; database; Ruby; Ruby on Rails; web browser; MVC



**Citation:** Klochkov, D.; Mulawka, J. Improving Ruby on Rails-Based Web Application Performance. *Information* **2021**, *12*, 319. https://doi.org/10.3390/info12080319

Academic Editor: Carlos Filipe Da Silva Portela

Received: 1 July 2021
Accepted: 3 August 2021
Published: 9 August 2021



## 1. Introduction

Web applications and services play a prominent role in our society, affecting numerous aspects of our day-to-day lives. Provided this prominent role, the possible cost of various flaws in web applications—unresponsive services, unintuitive user interfaces or unsecure systems—are also historically high. The importance of application performance, manifested in the page response time, or the perceived page response time, in particular, cannot be understated. Web users expect a page to load within two seconds, with every consecutive second critically lowering the quality of user experience and resulting in about a 10% increase of users abandoning the page per second [1].

A web application framework is a tool created to facilitate the process of web application development, define the basic structure and functionality of the web applications and make such flaws less likely to occur as the effect. In this contribution, we mainly focus on one web framework in particular—Ruby on Rails (RoR) [2]. This approach is based on

the dynamic multi-paradigm Ruby programming language. While being one of the first frameworks of its kind and having an immense influence on key aspects of web application development, RoR remains prominent to this day. However, RoR has ceded ground to its competitors in recent years and is no longer the most popular backend framework [3], with one of the speculative reasons for this development being relatively slower performance of Rails-based applications compared to other frameworks, such as Django, Laravel, Flask or Express [4].

The primary objective of this paper is to show the following: By using the API-only subset version of Rails framework with various improvements in conjunction with moving to client-side rendering and Single Page Application (SPA)-like architecture, it is possible to achieve substantive improvements in application page response time, without the downsides of reduced functionality or increase in code complexity.

## 2. Web Application Request Handling

The understanding of each step that occurs on a web application server and in browser between the instance of time when a user navigates to a particular page to the moment the page is displayed in browser and is interactive, is crucial for the ability to comprehend and optimize web application performance. In Figure 1, a general overview of the application request handling process of a Rails application is given. Even though, the elements and the code are typical for Rails, most entities and steps of this process are similar in other relevant Model-View-Controller (MVC) frameworks.

The initial step is the user request. Based on the manner of interaction of the end-user with the web page, two different kinds of requests can be differentiated:

- Explicit requests are directly initiated by the user. For example, those requests occur after the user has clicked the link or entered the Uniform Resource Locator (URL) into their browser's address bar. Commonly associated with the GET HTTP request method.
- Implicit requests occur in the background, oftentimes asynchronously, without the direct knowledge of a user and are indirect result of user actions. For example, different tracking scripts can initiate requests of this type. Commonly associated with the POST HTTP request method.

If the host navigates to a page for the first time, the Domain Name Server (DNS) lookup takes place as a result of a request. The Transmission Control Protocol (TCP) handshake as well as Transport Layers Security (TLS) key negotiation are other steps that take place if the connection was not established prior to the given request [5]. Upon the application server registering the request, it has to establish which particular application module or functionality is being accessed. One approach to this is to map combinations of fragments of the URL with generic parameters and HTTP action-verbs (POST, GET, DELETE, etc.) onto different controller 'actions', represented by functions in a controller class. After establishing the controller 'action', the body of a relevant controller function starts executing. Typically, for the sake of security, the first step of the controller routine is checking whether the user is authenticated and has the rights to perform the given action, and whether the session is not expired, and relevant security tokens are valid. After this step, the retrieval of relevant data from the database typically takes place, with the further possibility of modifying or deleting retrieved data from the system and the database.

Since the objects or entities in the system are represented by Objects (in the Object-Oriented Programming sense), while the databases which store the information about those objects typically conform to Relational model, with those two models being similar but not necessarily one-to-one congruent, the intermediary layer between the Model and the database needs to be included. The Object-Relational Mapping (ORM) is used in web application frameworks for this purpose. ORM maps the class methods of the model onto commands in an appropriate Data Manipulation Language of a given Database Management System (DBMS). The retrieved data is stored as an instance of a given Model or Object. In case the instance has been modified, it can then be written back to the database,

to maintain Consistency and Durability (in the ACID sense) of the application system and the database.

Each controller routine typically ends with a render command. Server-side rendering typically involves inserting the data from the retrieved Model instances into the View. To make this process easier and more developer-friendly, most frameworks utilize the Templating languages, which combine HTML syntax and tags with a subset of various instructions of a scripting language (those instructions most commonly consist of conditional statements and loops). The final result of this stage is a fragment of HTML document in which information about the objects of interest is inserted into the HTML tags.

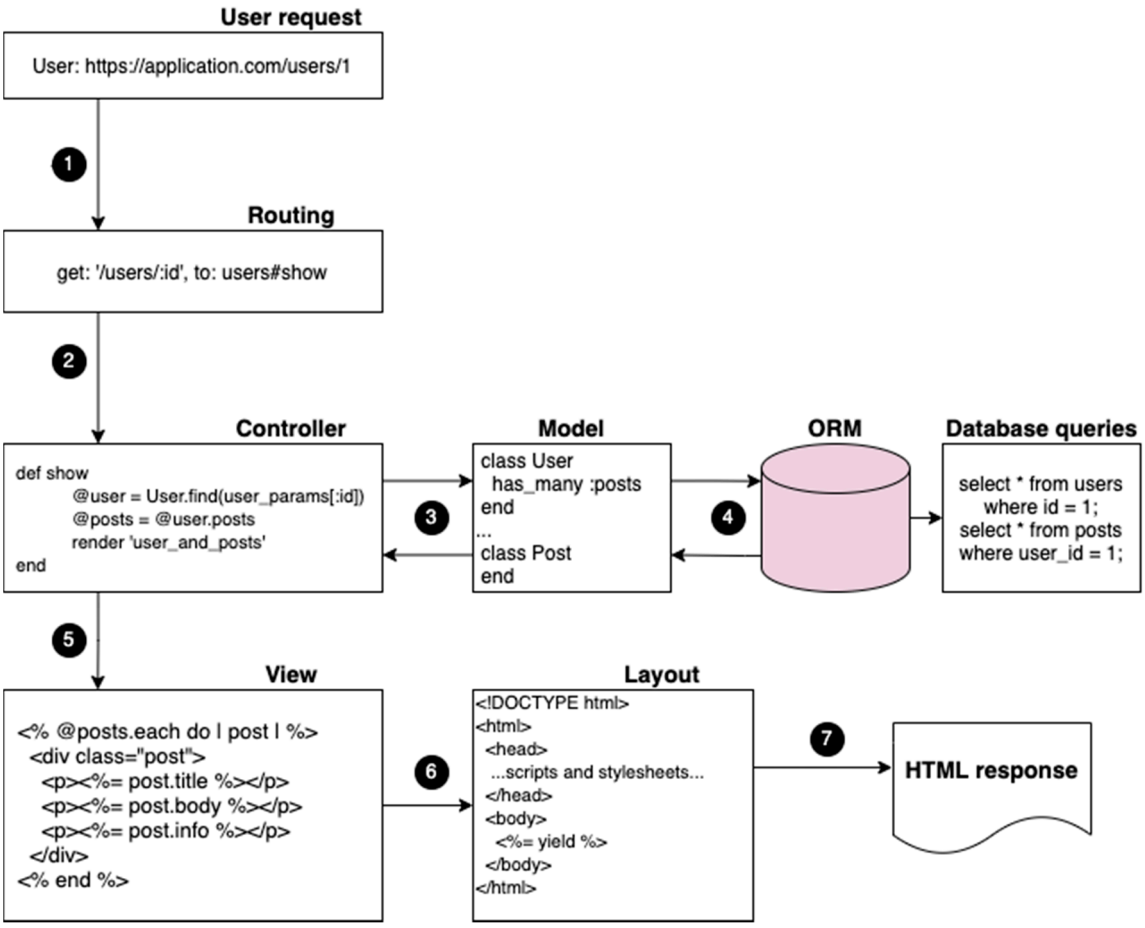


**Figure 1.** Web application server request handling process in a traditional Rails Application.

One of the final steps is assembling the proper HTML document, by combining the fragments with the necessary meta information, style and script tags, typically present inside the HEAD section of the HTML document. The Layout application element typically defines the way an HTML page is assembled. It also describes the layout-forming visual and functional elements of the application, that are common between multiple pages of an application, such as the navigation bar, side bars, headers, footers, etc.

Ultimately, the prepared HTML document is being sent back to the user. Once the first parts of the document have been received, the browser begins processing it. The process of turning the HTML document received from a server into an interactable page on the screen of a device, known as the critical rendering path [5], can be divided into the following steps:

1. The HTML elements are being parsed into the DOM (Document Object Model) Tree;
2. In a similar process, the CSSOM Tree arises from the CSS (Cascading Style Sheets) rules specified in the style sheets included in the document;

3. Both the CSSOM and DOM Trees are then combined into the Render, or Style Tree, on the basis of which the computed styles, applied to each visible page element, are established;
4. Render Tree information is then used to establish the position, height and width of each element, called the Layout;
5. The final step, called Painting, involves painting the individual nodes as pixels on the screen.

Once the painting step has been finished and all the blocking scripts included in the document have finished executing, the page becomes interactive.

The process of assembling the HTML page on the application server described above is also known as Server-Side rendering, and the applications based on this process will be referred to as 'monolith' applications (due to the entirety of codebase being in the one framework), as opposed to Client-Side Rendering applications, which have a clear divide between the Server-side data and logic layer and Client-side View layer.

## 3. Ruby on Rails Performance Considerations

RoR [2] is an open-source web application development framework. By building upon the assumptions and idioms that have emerged in the web development community, it is designed to make web development easier and more straightforward. It draws heavily upon the Model-View-Controller (MVC) architectural pattern and has two self-ascribed guiding core principles:

The Don't Repeat Yourself (DRY) principle emphasizes the tendency to avoid repeating the same knowledge (can be broadly understood as data, algorithms, functions, etc.) multiple times at various places of the system. Instead, the repeated functionality can be relegated to the special components (Helpers, Concerns, Mixins), that can be included into the functional parts of the applications to enable shared functionality without code repetition.

The Convention over Configuration (COC) principle means that Rails can be described as an 'opinionated' software that comes with certain preconceptions about the core architecture and functionality of an application. Those conventions enable the quick development of the application but can be overridden if more custom behavior is required [2].

Ruby on Rails has been the primary factor behind the popularity of Ruby programming language and those two technologies have become tightly linked together to the point of codependency.

### *3.1. Caching*

In a general sense, caching is the concept that involves storing the result of an operation so that as long as the result is not 'expired', the repeated invocations of said operation are expected to have the same results, thus, instead of performing all steps of an operation, it is sufficient to just return to this stored result. Caching is present on multiple levels of both software and hardware. Modern CPUs have multiple cache memories, the SSD storages use flash caching to achieve greater access speed. On a software level, multiple DBMS systems support function result caching.

In the context of web development and web applications, caching means that entire HTTP responses, HTML documents or fragments of documents can be stored and served to the users upon repeated requests. Oftentimes, the execution of the entire request handling process can be avoided and replaced with the check to determine whether the requested page is present in the cache: If so, the next step is to establish whether the cache entry has 'expired'. Oftentimes, this check comes down to finding out if any of the relevant objects have been modified or new related objects have been created since the time the page was last generated.

Rails provides the timestamps for the modification and creation of system objects. Cache keys as well as partial View caching can be used to further optimize caching behavior. Intelligent caching can be one of the most effective ways to boost application performance [6].

### *3.2. Database Access*

Since every datapoint that ends up on a web application page is retrieved from the database at a certain point, slow database accesses can be another performance bottleneck of an application. It is important to be cognizant of the fact that each tag with object data on a page has a certain performance cost [7]. So, including only relevant information on the page may not only be a good user experience guideline, but also an important consideration in terms of application performance.

It is important to selectively retrieve only the necessary data from the database, ideally only at times when it is necessary [8]. Since ORM is not a perfect system, a developer needs to be aware of certain performance pitfalls when it comes to object representations of relations. One such common pitfall is the N + 1 query problem presented in Figure 2a.

```
users = User.limit(10)
users.each do |user|
puts user.posts.inspect
end
```

(**a**)

```
users = User.includes(:posts).
users.each do |user|
puts user.posts.inspect
end
```

(**b**)

**Figure 2.** (**a**): Example of N + 1 Query Problem. (**b**): Resolving the N + 1 Query Problem using the includes() function.

The execution of the code presented in Figure 2a will result in 11 database queries: One to retrieve the first 10 users, and another one for each user to retrieve their posts based on their id. Because the practice of iterating over all objects that belong to different 'owner'-objects is quite common in web development, the function includes(), which allows it to specify the relationships to be included in the result set and is implemented in the core Rails framework.

This function, showed in Figure 2b, allows to achieve a substantive performance boost compared to the first example, by decreasing the number of database queries needed to execute the code only to 2: First query is executed to retrieve the list of user ids, and the second one mass-retrieves posts that belong to users on the id list [9].

### *3.3. RoR Modules*

Since Rails is a general-purpose framework, it includes a set of Components, Modules and Middleware that can be utilized to develop and maintain different types of applications [10]. Some of those modules are redundant in certain applications and can be disabled. In particular, most of the modules and middleware can be disabled for API-only applications. Such a configuration was once a part of the community-created custom Ruby on Rails library, called 'gem' [11], but it has since become a part of the core Rails framework [11,12] and can be used to create an application that uses a subset of Rails framework that offers only basic functionality of routing requests, accessing the database and ORM, and responding with data in a serialized, typically JSON, format.

## 4. Software Implementation

In order to evaluate the impact of different types of optimizations on application performance, the benchmark application was implemented. This is a simplified version of modern feed-driven social networking application. The application has multiple endpoints that differ in terms of database request intensity as well as page elements count. Test endpoints are presented in Table 1.

**Table 1.** Specification and general description of benchmark application test endpoints.

| Endpoint | Database Load | Page Element Count | Description |
| --- | --- | --- | --- |
| /feed | Very High. Every table in the database schema is involved. High number of resource-intensive join queries are executed. | High. For each of N posts, about 250 HTML elements are present. Total element number varies depending on the number of posts on a page. | Displays the timeline of posts either authored by the current user or by other users that are followed by the current user. For each post, comments are shown as well. The display of every post or comment also includes the information about the author or respective content as well as number of upvotes and display of number of reactions of different types. Information about the author of the content includes number of following and followed users, number of posts as well as rating, defined as sum of upvotes of every post authored by a user. |
| /post/:id | Medium. Multiple tables in the database schema are involved. A few resource-intensive join queries are executed. | Low. Total page HTML element count is about 300. | Displays a single post along with relevant information, as described in the feed endpoint. |
| /user/:id/followers | Low. A few tables in the database schema are involved. Single resource-intensive join query is executed. | Moderate. For every user, about 50 HTML elements are present. Total element number less than 2000. | Displays the table that contains all of the followers of the current user. The information of each follower is similar to that described in the feed endpoint. |

In order to evaluate the impact of different types of optimizations on application performance, the benchmark application was implemented.

The schema of the database used in the application server is presented in Figure 3. Among notable things, the password digest needed for user authentication is stored in the Users table, the exclusive OR relation between the Reaction and Post or Comment entities is resolved by adding content_type and content_id columns to the Reactions table, which corresponds to a polymorphic association on the ORM level in the application. The self-referential relations between Users, Users-Followers and User-Followees are modeled by adding an additional table that stores pairs of follower and followee ids.

After implementing baseline application as a monolithic Ruby on Rails application that uses Server-Side Rendering, a sequence of steps was taken in order to bring down the page response time and improve the user experience. The steps can be divided into two categories: Backend Optimizations and Rendering Strategy and Frontend Optimizations. After each step, application performance measurements were taken.

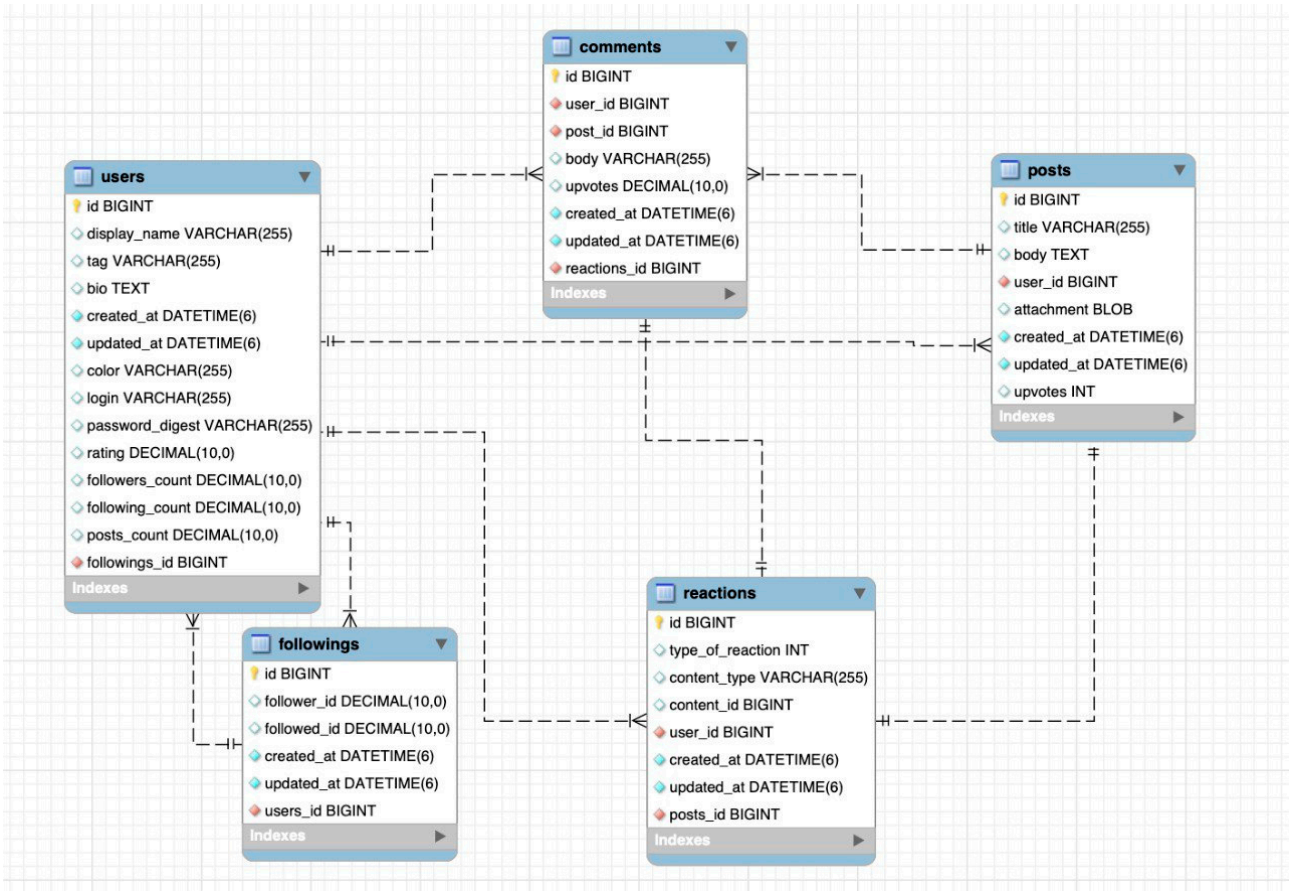


**Figure 3.** The overview of the application database schema.

### *4.1. Rendering Strategy: Transition towards Client-Side Rendering*

In order to make a transition from a monolithic web application towards a client-side rendering, or Single Page Application architecture in particular, the following fundamental adjustments need to be made [13,14]:

- The elements of View layer—Templates and Layouts need to be remade to take the input data either in the form of Javascript Objects or JSON notation and output HTML elements. In other words, the templating mechanism needs to be reimplemented in the JavaScript-compliant technology.
- Controllers need to be functionally divided into two categories: API Controllers, the primary role of which is to respond to requests from the frontend with the appropriate serialized data, and Pages, or Entry-Points Controllers, the role of which is to serve usually pre-determined and static at the point of server-side generation, resources, usually HTML documents with the included frontend scripts that are responsible for the frontend logic. For SPA, only a single such page needs to be provided at the first load of the application. The latter category of controllers is largely optional, as another parallel gateway application can be used instead [15].
- Routing and the state-management needs to be re-implemented in the Javascript frontend.
- Serializers—Components that transform the information about the attributes of relevant Model layer objects and associations between them to a standardized form, typically JSON or XML, need to be implemented [16,17].

Most Model components, responsible for business logic and data representation, remain largely unchanged between the studied monolith and SPA applications. In addition, the logic and principles underlying the Templating mechanism, Routing, State management and other components present in both types of architecture are also overlapping to a large

extent, as demonstrated by the example of the custom templating mechanism in JavaScript (Figure 4a) compared to an HTML templating language Slim [18], popular in monolith Rails applications (Figure 4b).

```
.post-container href="/post/#{post.id}"
  a.user-container href="/user/#{post.user.id}"
    ==render partial: "shared/user_mini", locals: {user: post.user}
  .text-container
    .post-title = post.title
    .body-container
      .body = post.body
    .additional-block.info
      .upvotes-block
        .btn-vote.up
        .upvote.count = post.upvotes
        .btn-vote.down
      .reactions-block
        -post.non_vote_reactions.group(:type_of_reaction).count.each do |type,count|
          .emoji-block
            span.count = count
            span.emoji = Reaction.get_emoji(type.to_sym)
        .emoji-block.plus-block
          span.emoji ="+"
      .comments-total-block
        span Comments:
        span.count = post.comments.count
    .comments-wrapper
      ==render partial: "comments/show", collection: post.comments, as: :comment
```

(**a**)

```
var view = `
  <div class="post-container" data-link href="/post/${this.post.id}">
    <div class="user-container">
      </user-block>
    </div>
    <div class="text-container">
      <div class="post-title">${this.post.attributes.title}</div>
      <div class="body-container">
        <div class="body">${this.post.attributes.body}</div>
      </div>
      <div class="additional-block info">
        <div class="upvotes-block">
          <div class="btn-vote up"></div>
          <div class="upvote count">>${this.post.attributes.upvotes}</div>
          <div class="btn-vote down"></div>
        </div>
        <div class="reactions-block">
          </content-reactions>
          <div class="emoji-block plus-block"><span class="emoji">+</span></div>
        </div>
        <div class="comments-total-block">
          <span>Comments:</span>
          <span class="count">${this.post.relationships.comments.data.length}</span>
        </div>
      </div>
      <div class="comments-wrapper">
        </comments>
      </div>
    </div>
  </div>
`
```

(**b**)

**Figure 4.** (**a**): Templating mechanism for a Post View implemented with the Slim templating language. (**b**): Templating mechanism for a Post View implemented with JavaScript.

### *4.2. Backend Optimization*

#### 4.2.1. Resolving the N + 1 Query Problem and Preloading Optimizations

Feed endpoint uses data from multiple interconnected database tables. In the code displayed in Figure 5, the code responsible for retrieving the posts in the feed, and only posts either authored by the current user or his followed users are selected.

```
followed_ids = @user.followed.collect(&:id)
followed_ids << @user.id
@posts = Post.where(user_id: followed_ids)
        .order(updated_at: :desc).limit(POST_LIMIT)
```

**Figure 5.** Section of Feed Controller that retrieves relevant posts.

Since for each retrieved post, multiple database queries regarding posts' comments and the users-authors of those respective comments will be fired consecutively. The code listed in Figure 5 can be seen as an instance of N + 1 query problem.

The Rails framework provides multiple methods that could alleviate this issue, by preloading information from the tables associated with the given class. To this end, Rails provides four different methods [19]:

1. Joins()—Executes a single database query that uses INNER JOIN;
2. Includes()—Loads data from the associated table by either executing a separate query or using the single LEFT OUTER JOIN query, depending on whether the WHERE condition was given for the associated table;
3. Preload()—Similar to Includes(), but always executes a separate query;
4. Eager_load()—Similar to Includes(), but always executes a single LEFT OUTER JOIN query.

As an intermediary step in the backend optimization process, the performance metrics of different preloading functions were compared. A select few configurations are given in Table 2. For this case, since no additional condition block is given (every single comment of a post needs to be included), the Preload() is functionally equal to the Includes() method

**Table 2.** Comparison of different preloading functions for the Feed (100) endpoint.

| Preloading Function | View Time (ms) | Database Time (ms) | Total Time (ms) | # of sql Queries Executed |
|---|---|---|---|---|
| (1) Baseline (no preloading) | 2410.5 | 863.1 | 3273.6 | 1423 |
| (2) incudes (:comments, comments: :user) | 2016.0 | 752.3 | 2768.3 | 1189 |
| (3) eager_load (:comments, comments: :user) | 2235.2 | 825.1 | 3060.3 | 1197 |
| (4) includes (:reactions, :comments, comments: :user, comments: :reactions) | 2495.0 | 855.5 | 3350.5 | 1317 |
| (5) joins (:reactions, :comments, comments: :user, comments: :reactions) | 4094.1 | 185.7 | 4279.8 | 167 |

Out of all the different tested combinations of preloading functions and associations, example (2) was experimentally established to be the best performing combination for this endpoint.

As exemplified by example (5), preloading a large amount of data might free up the database, due to a smaller number of queries fired, but the huge chunks of the retrieved data need to then be manipulated on the View layer in order to be properly filled into the resulting page, which causes a drastic decrease in the overall performance, however, such an approach might be a good choice when the database resources are severely limited.

#### 4.2.2. Using the Rails API-Only Subset

Rails provides a set of middleware for general purpose applications by default. It is possible to disable redundant parts of this middleware when using the "api-only" configu-

ration. Among the disabled middleware are different modules responsible for assembling and delivering the appropriate HTML or other format responses to user requests, storing the session state in cookies and others. When this configuration is selected, however, the serving of regular HTML responses becomes not supported, and the application cannot function properly. In order to overcome this issue, a "regular" Entry-Point controller was created by overriding the default behavior of API-only controllers, with the sole task of serving as the default index page with the relevant scripts and stylesheets (see Figure 6).

```
require 'rails/application_controller'
class EntryPointController < Rails::ApplicationController
def index
render file: Rails.root.join('public/index.html')
end
end
```

**Figure 6.** EntryPoint Controller in Rails API application.

#### 4.2.3. Database Access Improvements

The user information displayed on both "feed" and "followers" endpoints contains, among others, the information about the number of user posts, user followers as well as following accounts, and rating, which is a weighted sum of the score of every piece of content a user has published (both posts and comments). Originally, an aggregate query was executed in order to retrieve this information for each user. This approach turned out to be suboptimal and resulted in a lot of excessive database queries executed for each endpoint. In order to alleviate this problem, four further solutions were evaluated:

1. Baseline: When a comment or a post is upvoted or downvoted by a user, the number of its upvotes is updated in the database. This, however, does not affect the rating of a user who published the content. The aggregate query is executed each time to retrieve the rating.
2. Caching: The number of upvotes is stored in the application cache, when the upvote is registered, the rating cache of the author-user becomes invalidated.
3. Database storage: The user rating is stored in the database and is retrieved each time; upvotes will result in updating.
4. Database storage and caching: Combination of two above approaches; the database query is only executed if the cache is not valid.

Those four approaches were compared by executing a series of 1000 typical operations and evaluating the results. The two operations were: "fetch + get" (retrieving the user and the information about his rating) and "update + fetch + get" (updating the post's upvotes, which will result in change of user's rating and then retrieving user and information about his rating). The results are displayed in Table 3.

The optimal approach for this case was determined to be the (3) Database, since the time to retrieve the User rating is the lowest in the "fetch + get" scenario (this is explained by the fact that after retrieving the User information from the database, no additional operations are required to obtain the rating of a user). Contrary to the original conception, the Caching solution would be preferable if the User rating was updated very frequently. This is due to the fact that updating a post's or a comment's upvotes does not result in an additional database writing operation in this case, and the cost of invalidating the application's cache is negligible. The combination of both Caching and Database storing can be a good middle ground between those solutions, especially if the cached database operation was very costly, but in this case, it is not preferable to either of them.

**Table 3.** Comparison of database storing and caching of User rating, time for 1000 operations.

| Storing and Retrieval Strategy | "Fetch + Get" | "Update + Fetch + Get" |
|---|---|---|
| (1) Baseline | 1388 ms | 5477 ms |
| (2) Cache | 479 ms | 5457 ms |
| (3) Database | 355 ms | 6107 ms |
| (4) Database + Cache | 460 ms | 6361 ms |

#### 4.2.4. Introducing Caching to Serializers

Object Serializers can be typically invoked multiple times per single API call (i.e., when a single user has authored multiple posts or comments on a feed page, Author Serializer can be invoked once per each post or comment). In addition, typical serializer procedure combines retrieving information about associated objects from the database with the transformation of this information into a standardized format, which adds computational overhead. Thus, a serializer appears to be an optimal place for implementing application caching, especially in case when a large percentage of object retrievals result in the call of corresponding serializers.

The Fast JSONAPI gem provides a possibility of including caching directly in the serializers. Instead of performing the serializer routine, the presence of a non-expired cache entry is checked first. The procedure is only performed if such entry is not found. Different cache intervals as well as custom behavior in case of cache miss can be defined [17].

## 5. Performance Measurements

Performance measurements were performed with the rails_performance Gem [20] in the controlled local network environment, with a single user at a time. After each optimization step, a series of performance measurements were taken for four configurations: "/feed" endpoint with number of posts N = 100 and N = 10, "/user/:id/followers" endpoint. In addition, a more extensive series of tests were performed to establish different performance metrics of baseline and final optimized application configuration. Measurements for each step are listed in Table 4. The rails_performance Gem, which provides data for the amounts of time spent in the View and Database (Model) layers, was used in conjunction with the Rails' internal profiling capabilities. Furthermore, the Google Chrome browser's built-in developer measuring tools were utilized, to provide the total time until the page was ready as well as the time to the initial load [21].

Performance measurements results are given in Table 4. For the baseline application and after each step, the following metrics were taken:

- Time (VW)—Time spent in the View Layer of the application, directly obtained from the Rails internal monitoring for monolith applications and calculated as the difference between the moment the page was initially received and the moment when the page was fully rendered in Client-Side Rendering applications (marked with **);
- Time (DB)—Time spent in the Model Layer of the application, taken from the Rails monitoring;
- Time (DOM)—Time from the first load of the page until the page was fully rendered and interactive in the browser, measured with the use of Google Chrome's profiling tool.

### *5.1. Results Analysis*

Five discrete optimization steps were taken:

1. First was Resolving the N + 1 Query problem and preloading optimizations (marked as N + 1&Pre). After this step, due to more optimal resource loading and management, both Database Time and View Time decreased as predicted, and the page loading time decreased as well. This effect was most pronounced for the Feed (100) endpoint due to the large amount of data retrieved.

2. Second was switching to the Client-Side Rendering strategy. This resulted in an overall decrease of Feed endpoint page loading time; however, due to another metric used for View Time, there appears to be an increase in the amount of time spent in the View layer, but it is not conclusive.
3. Another step was using the API-only version of the Rails framework. This resulted in an overall increased efficiency of both communication with the database and serialization process, included in the Time (VW), which resulted in somewhat better page loading times.
4. The fourth step was optimizing database accesses. This resulted in the most drastic increase in the database communication efficiency. Because of the improved performance of the object-level serializers, the View Time appears to have decreased sharply as well. A very data intensive Feed (100) endpoint has improved the most.
5. The fifth step was introduction of the Object-level serializer caching. The Database Access time plummeted across all the endpoints due to only initial database access being necessary in order to produce a serialized object. Predictably, the most data-heavy endpoint has benefited the most, in part due to high degree of object interrelations. The least data intensive endpoint has benefited the least, due to the smaller time difference between retrieving the serialized object from the cache and building the same object based on information accessed from the database. In other words, the gain from serializer caching was highly correlated to the degree of object interconnectedness as well as the regular serializer process overhead.

Due to the drastically different dataflow processes in the last application configuration, that involved serializer Caching, the previous configuration was chosen for the further benchmarks instead.

### *5.2. Comparison with the Baseline Application*

A more extensive series of tests was performed to compare the baseline and the optimized versions of the application. In order to maintain the continuity in dataflow between the baseline and optimized version of the application, the non-cached version of the optimized application, presented in the fourth step of the optimization, is chosen for those benchmarks. Firstly, the initial load times of all the implemented endpoints were compared, as can be seen in the Figure 7.

In the graph in Figure 7, the entire bar represents the page loading time (or DOM time). It is divided into two parts—the Server Response Time, defined as the time from the start of a request until the application server has finalized and sent its response, and the Rendering time, which is the difference between the DOM time and the server response time. As is the case with the other optimizations, the effect is most pronounced, having the most data and page element intensive endpoint out of the three, that is, Feed (10). Among notable things, not only has the server response time decreased sharply, the Rendering time appears to also have fallen, which is due to the JSON payload (used in the optimized version of the application) taking less space and thus travelling faster through the network. The browser also does not need to spend a lot of time parsing a heavy HTML, and this process is overall faster because JavaScript DOM manipulation capabilities are used instead.

Those conclusions are further substantiated by the relationship shown in Figure 8: The difference between the Page Response Time and the Server Response Time for the optimized applications is virtually constant for the Client-Side Rendering application. At the same time, this gap seems to increase as the amount of the elements on HTML response page increases. This demonstrates the overhead time spent parsing the HTML in browser. Another factor is the Response Sizes of HTML (baseline application) and JSON (optimized). Both lines have a linear character, but the pale green line representing JSON is less steep and is lower than the HTML size line (this can be viewed as monolith applications being less data-efficient and taking up more network capacity to serve the same page).

**Table 4.** Endpoint performance measurements.

| Optimization Step | Endpoint Response Times | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Feed (100) | | | Feed (10) | | | Followers (30) | | |
| | Time (VW)/Δ% | Time (DB)/Δ% | Time (DOM)/Δ% | Time (VW)/Δ% | Time (DB)/Δ% | Time (DOM)/Δ% | Time (VW)/Δ% | Time (DB)/Δ% | Time (DOM)/Δ% |
| Baseline | 2326 | 844.8 | 4176 | 435.16 | 211.62 | 1240 | 176.42 | 107.34 | 789.6 |
| Δ% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% |
| N + 1&Pre | 2162.7 | 801.14 | 3910 | 428.68 | 208.3 | 1202 | N/A * | N/A * | N/A * |
| Δ% | 92.97% | 94.83% | 93.63% | 98.51% | 98.43% | 96.93% | N/A * | N/A * | N/A * |
| Client Side ** | 2767.2 | 790 | 2988 | 918.2 | 216.74 | 1126 | 604.6 | 123.66 | 817.4 |
| Δ% | 127.94% | 98.60% | 76.41% | 214.19% | 104.05% | 93.67% | 342.70% | 115.20% | 103.52% |
| API Only ** | 2632 | 771.42 | 2804 | 787 | 216.3 | 933.8 | 551.8 | 119.8 | 705.6 |
| Δ% | 95.11% | 97.64% | 93.84% | 85.71% | 99.79% | 82.93% | 91.26% | 96.87% | 86.32% |
| DB opt ** | 1557.2 | 270.34 | 1756 | 545.4 | 87.28 | 724.4 | 380.2 | 45.76 | 592.4 |
| Δ% | 59.16% | 35.04% | 62.62% | 69.30% | 40.35% | 77.57% | 68.90% | 38.19% | 83.95% |
| Srl. Cache ** (Cache H.R = 100) | 571.2 | 41.28 | 822.4 | 389.6 | 23.02 | 651 | 339.6 | 5 | 556 |
| Δ% | 36.68% | 15.26% | 46.83% | 71.43% | 26.37% | 89.86% | 89.32% | 10.92% | 93.85% |

* For Followers endpoint, Preloading was not utilized, ** Different View Time measurements for SPA-type applications, "Δ%" Rows mark the relative difference between the current results and results of the previous step. Cache H.R.—Cache Hit Ratio, chosen to be 100% for the experiment, the approximation for intermediary values can be calculated as the weighted sum of DB opt and Srl Cache values.

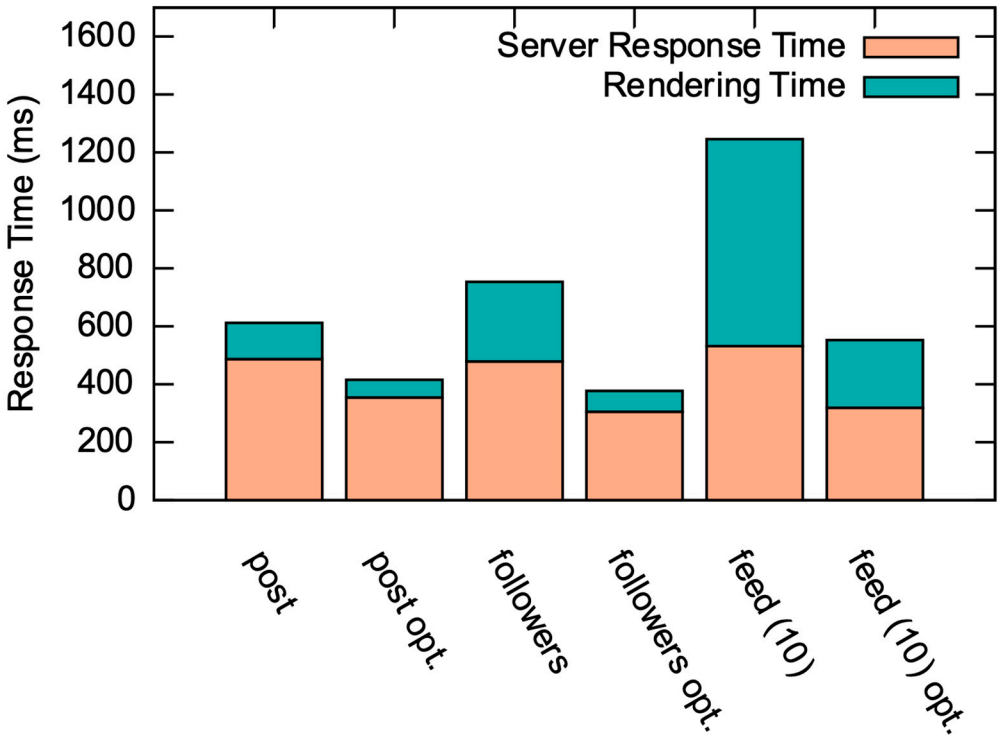


**Figure 7.** Time metrics of implemented endpoints between the optimized and baseline applications.

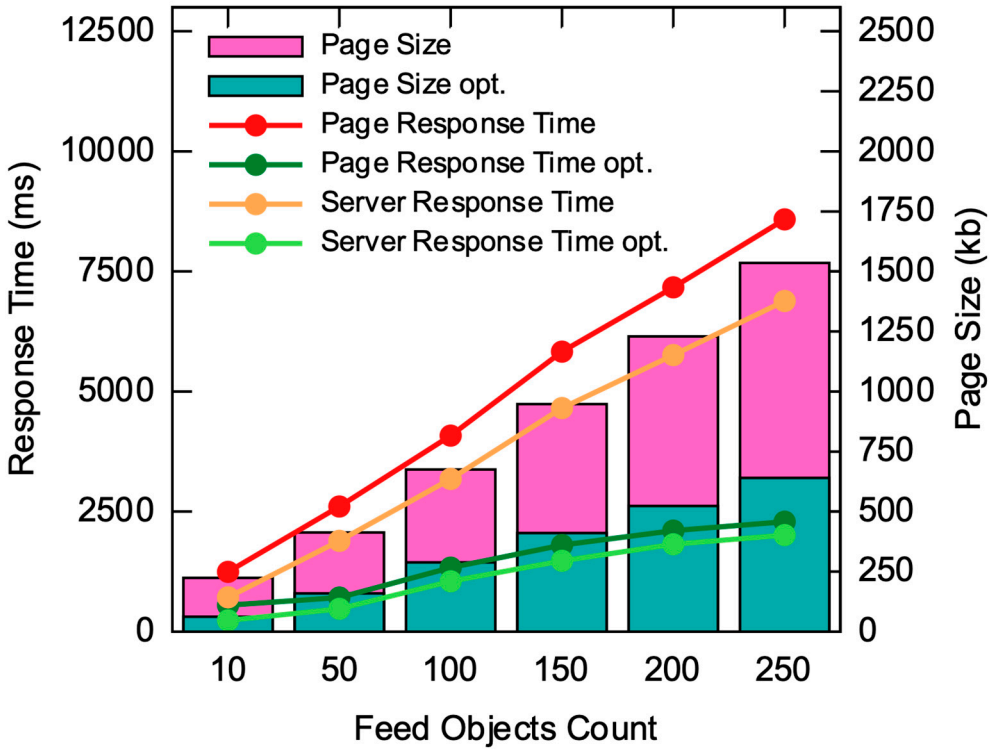


**Figure 8.** Relationship between the number of objects retrieved, the response time (lines) and the page size (bars) for optimized and baseline applications.

## 6. Conclusions

The main goal of this article—to quantitatively and qualitatively describe the steps of the optimization of a Rails-based application—was achieved. The initial load response times of the final optimized application are up to 74% faster compared to the baseline application. Different aspects of the application's performance as well as avenues of optimizations were explored during the course of this work. This article opens up different possibilities for further work and research in this field. The conclusions regarding the optimization steps can be applied to the existing applications, in particular, A-B testing can be undertaken to see both the effects of different optimizations on the response times of existing web applications, as well as the relationship between the application load times and the positive outcomes (financial, in the case of web-enterprise applications, and social, in case of government service) associated with a web application can be established in a similar manner. For Ruby on Rails applications, further work in this field with a possibility to build a universal tool that would help developers better monitor, recognize problems and improve performance can be undertaken. Despite the rather narrow scope and inherent limitations in this research, the conclusions can be applied to different web applications and web application frameworks.

**Author Contributions:** Both authors contributed equally to (conceptualization and writing) the manuscript. Both authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** All datasets generated for this study are included in the article.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Why Does Speed Matter? Web-Dev. Available online: https://web.dev/why-speed-matters/ (accessed on 21 June 2021).
2. Ruby on Rails Official Page. Available online: https://rubyonrails.org/ (accessed on 21 June 2021).
3. Most Popular Backend Frameworks-2012/2021. Available online: https://statisticsanddata.org/data/most-popular-backend-frameworks-2012-2021/ (accessed on 21 June 2021).
4. Crawford, T.; Hussain, T. A comparison of server side scripting technologies. In Proceedings of the International Conference on Software Engineering Research and Practice (SERP 2017), Las Vegas, NV, USA, 17–20 July 2017; pp. 69–76.
5. Populating the Page: How Browsers Work-Web Performance MDN. Available online: https://developer.mozilla.org/en-US/docs/Web/Performance/How_browsers_work (accessed on 22 June 2021).
6. Caching with Rails: An Overview. Available online: https://guides.rubyonrails.org/caching_with_rails.html (accessed on 19 June 2021).
7. Yang, J.; Yan, C.; Wan, C.; Lu, S.; Cheung, A. View-Centric Performance Optimization for Database-Backed Web Applications. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019; pp. 994–1004. [CrossRef]
8. Persson, K. Optimizing Ruby on Rails for Performance and Scalability. Degree Project in Computer Science. Master's Thesis, KTH Royal Institute of Technology, Stockholm, Sweden, 2016.
9. Optimizing Database Performance in Rails, Heroku. Available online: https://blog.heroku.com/rails-database-optimization (accessed on 21 June 2021).
10. Nordén, M. To What Extent Does Ruby on Rails Effect Performance in Applications. Degree project in Computer Science. Master's Thesis, Linköping University, Linköping, Sweden, 2015.
11. Ruby on Rails API Gem, GitHub. Available online: https://github.com/rails-api/rails-api (accessed on 25 June 2021).
12. Using Rails for API-only Applications, Rails Guides. Available online: https://guides.rubyonrails.org/api_app.html (accessed on 25 June 2021).
13. Mukhamadiev, A. Transitioning from Server-Side to Client-Side Rendering of the Web-Based User Interface: A Performance Perspective. Degree Project in Media Engineering. Bachelor's Thesis, Metropolia University of Applied Sciences, Helsinki, Finland, 2018.
14. Risto, O. A Performance Comparison of Rendering Strategies in Open-Source Web Frontend Frameworks. Degree project in Computer Science. Master's Thesis, University of Helsinki, Helsinki, Finland, 2021.

15. Shklar, L.; Rosen, R. *Web Application Architecture: Principles, Protocols and Practices*, 2nd ed.; Wiley Publishing: Hoboken, NJ, USA, 2009.
16. Marrs, T. *JSON at Work: Practical Data Integration for the Web*, 1st ed.; O'Reilly Media: Newton, MA, USA, 2017.
17. Fast JSONAPI Ruby on Rails Serializer Gem, GitHub. Available online: https://github.com/jsonapi-serializer/jsonapi-serializer (accessed on 26 June 2021).
18. Slim—A Fast, Lightweight Template Engine for Ruby, Slim Lang. Available online: http://slim-lang.com/ (accessed on 24 June 2021).
19. Rails API Reference, Rails API Dock. Available online: https://apidock.com/rails/ActiveRecord/QueryMethods/ (accessed on 27 June 2021).
20. Rails Performance Gem, GitHub. Available online: https://github.com/igorkasyanchuk/rails_performance (accessed on 27 July 2021).
21. Timeline Event Reference, Google Developers. Available online: https://developer.chrome.com/docs/devtools/evaluate-performance (accessed on 27 June 2021).