Aishwarya Iyer - ai2336
Kavya Premkumar - kp2652
Rohit Gurunath - rg2997
Siri Haricharan - sh3451
Somdeep Dey - sd2988

WebWarriors - Final Report

What we have delivered and how the teaching staff can access it

The final delivery of our project is in the form of a Github repository as promised in the progress report at the following location :

https://github.com/siri47/WebWarriors

The total github repository comprises separate folders for each web application and for the tests we built.

External software Used

The following list of external software was incorporated or used in some capacity in our project:

- Apache Benchmark Tool
  https://httpd.apache.org/docs/2.4/programs/ab.html
- Python - Flask
  http://flask.pocoo.org/
- Ruby on Rails
  http://rubyonrails.org/
- PHP
  https://secure.php.net/
- Node js
  https://nodejs.org/en/
- MongoDb
  https://www.mongodb.com/download-center#community
- Composer
  https://getcomposer.org/

Who did what

We essentially carried out the construction of 4 web applications based on four distinct web application frameworks :

- Ruby on Rails : Rohit and Kavya
- Python using Flask : Siri and Aishwarya
- NodeJs : Somdeep and Aishwarya
- PHP : Somdeep and Siri
- Test Scripts : Kavya and Rohit

The essential division of work was decided on the following basis : we intentionally divided work in a way that each of us had an opportunity to work on frameworks we had minimal previous knowledge of so as to better meet the requirements for this project. It lead to an interesting perspective as each of us had to initialize the software process right from scratch for each framework : installation, developing familiarity with the coding environment, package and library management, challenges that we have covered in greater detail in later sections.

For each of the web apps, each of us provided a separate folder in the overall Git repository. The work involved in the construction of each web app included the following aspects :

- Setting up of the overall development environment on each platform.
- Basic connecting of each framework with a common database (the MongoDb NoSQl database engine - running locally on each machine).
- Basic web forms were constructed that were used as attack surfaces. (for instance for cross-side scripting attack). They necessarily met the minimal requirement of having input boxes and additionally relaying input, to better demonstrate some of the attacks.
- Scripts were built for the purpose of inserting and deleting large amounts of data from the underlying database.
- Routes were provided for uploading and downloading images.

- Additionally, the modules for test scripts incorporated basic CSRF attacks and HTTP requests made to routes provided by our different web applications.
- Server side execution modules were built into each web app by locating libraries available for the AES 128-bit encryption algorithm, that involves considerable server-side computation with high CPU time.
- We then ran test locally on our different frameworks, as well as between devices over the local network, and incorporated these performance measures.

Results or Findings

Using Apache BenchMark tool and test scripts that we wrote ,the following properties of the web application were tested :

1. Performance :

Performance of the frameworks were tested with respect to the following functions :

- Video rendering - we used apache test bench to render a 1 MB video file multiple times with and without concurrency. This would measure how well it can handle multiple requests and how efficiently it can server concurrent requests
- Image rendering - The same test mentioned above was carried out for an image of size 104.5 KB
- Server Side computation - The idea was to test the raw performance of the server-side programming language being used, which required a CPU-intensive operation without intervention from the OS. This prompted us to use AES 128 bit encryption in CBC mode for image files, repeated over 10,000 times. This helped us estimate the raw performance of the server-side programming language in terms of the amount of time required to finish encrypting and decrypting the data 10,000 times.
- Database Writes and Reads - we used a custom python script to send HTTP requests to the web server, that triggered 100 writes to and reads from mongodb.

Results:

1. Flask :

A. Video Retrieval

1.Concurrency Level: 1
Time taken for tests: 236.008 seconds
Complete requests: 1000
Total transferred: 1056052000 bytes
Time per request: 236.008 [ms] (mean, across all concurrent requests)

2. Concurrency Level: 10
Time taken for tests: 23.283 seconds
Complete requests: 100
Total transferred: 105605200 bytes
Time per request: 232.831 [ms] (mean, across all concurrent requests)

B. Image retrieval

1. Concurrency Level: 1
Time taken for tests: 44.985 seconds
Complete requests: 1000
Total transferred: 104499000 bytes
Time per request: 44.985 [ms] (mean, across all concurrent requests)

2. Concurrency Level: 10
Time taken for tests: 4.084 seconds
Complete requests: 100
Total transferred: 10449900 bytes
Time per request: 40.841 [ms] (mean, across all concurrent requests)

C. Image Encryption and Decryption

Time taken : 14 seconds
Complete requests: 1
Rounds of encryption/decryption: 10000

D. Database query

Complete requests: 1000
Time taken to write values = 0.767047 seconds
Time taken to read values = 0.085928 seconds

2. Ruby :

A.Video Rendering

1. Concurrency Level : 1
   Time taken for tests : 227.283 seconds
   Complete requests: 1000
   Total transferred: 1056202000 bytes
   Time per request: 227.283 [ms] (mean)
   Time per request: 227.283 [ms] (mean, across all concurrent requests)

2. Concurrency Level : 10
   Time taken for tests : 9.187 seconds
   Complete requests: 100
   Total transferred: 1056202000 bytes
   Time per request: 918.661 [ms] (mean)
   Time per request: 91.866 [ms] (mean, across all concurrent requests)

B. Image Rendering

1. Concurrency Level: 1
   Time taken for tests: 63.908 seconds
   Complete requests: 1000
   Total transferred: 104650000 bytes
   Time per request: 63.908 [ms] (mean)
   Time per request: 63.908 [ms] (mean, across all concurrent requests)

2. Concurrency Level: 10
   Time taken for tests: 1.015 seconds
   Complete requests: 100
   Total transferred: 104650000 bytes
   Time per request: 101.542 [ms] (mean)
   Time per request: 10.154 [ms] (mean, across all concurrent requests)

C. Server side computation (Encryption and Decryption of an image)

Time taken for tests: 159 seconds
Completed requests: 1
No of rounds of Encryption and Decryption: 10000

D. Storing data and retrieving from database

Complete requests: 1000
Time taken to write values = 6 seconds
Time taken to read values = 1 seconds

3. PHP :

A.Video Rendering

1. Concurrency Level : 1
   Time taken for tests : 227.283 seconds
   Complete requests: 1000
   Total transferred: 1056202000 bytes
   Time per request: 227.283 [ms] (mean)
   Time per request: 227.283 [ms] (mean, across all concurrent requests)

2. Concurrency Level : 10
   Time taken for tests : 9.187 seconds
   Complete requests: 100
   Total transferred: 1056202000 bytes

Time per request: 918.661 [ms] (mean)
Time per request: 91.866 [ms] (mean, across all concurrent requests)

B. Image Rendering

1. Concurrency Level: 1
   Time taken for tests: 63.908 seconds
   Complete requests: 1000
   Total transferred: 104650000 bytes
   Time per request: 63.908 [ms] (mean)
   Time per request: 63.908 [ms] (mean, across all concurrent requests)

2. Concurrency Level: 10
   Time taken for tests: 1.015 seconds
   Complete requests: 100
   Total transferred: 104650000 bytes
   Time per request: 101.542 [ms] (mean)
   Time per request: 10.154 [ms] (mean, across all concurrent requests)

C. Server side computation (Encryption and Decryption of an image)

Time taken for tests: 159 seconds
Completed requests: 1
No of rounds of Encryption and Decryption: 10000

D. Storing data and retrieving from database

Complete requests: 1000
Time taken to write values = 6 seconds
Time taken to read values = 1 seconds

4. NodeJS :

A.Video Rendering

1. Concurrency Level : 1
   Time taken for tests : 384 seconds
   Complete requests: 1000
   Total transferred: 1056202000 bytes
   Time per request: 384 [ms] (mean)
   Time per request: 384 [ms] (mean, across all concurrent requests)

B. Image Rendering

1. Concurrency Level: 1

Time taken for tests: 190 seconds
Complete requests: 1000
Total transferred: 104650000 bytes
Time per request: 190 [ms] (mean)
Time per request: 190 [ms] (mean, across all concurrent requests)

C. Server side computation (Encryption and Decryption of an image)

Time taken for tests: 671 seconds
Completed requests: 1
No of rounds of Encryption and Decryption: 10000

D. Storing data and retrieving from database

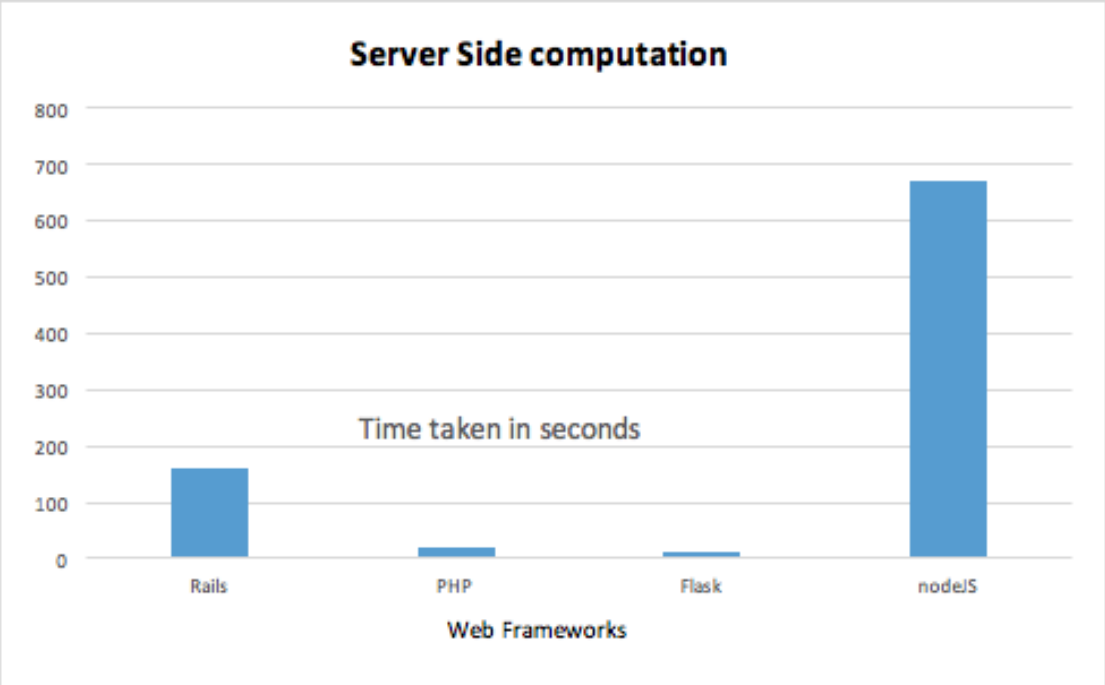
Complete requests: 1000
Time taken to write values = 0.402 seconds
Time taken to read values = 0.257 seconds

Measurements and Comparison

Server-side computation:

	Rails	PHP	Flask	nodeJS
Time taken	159	21	13	671

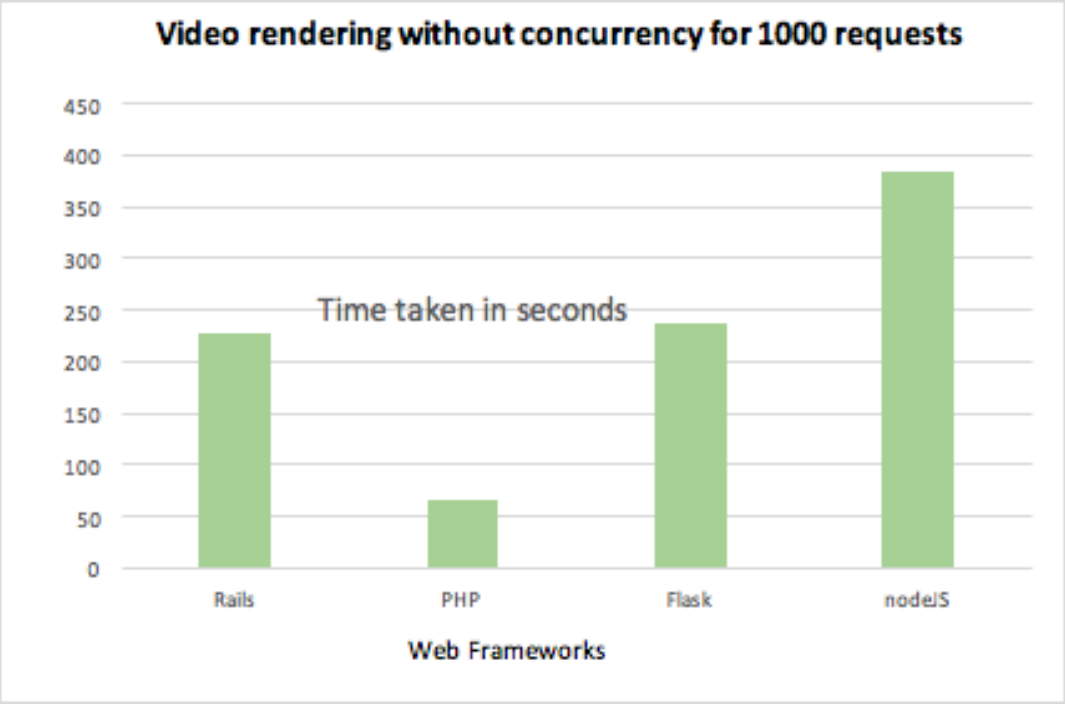
***time in seconds**



Video Rendering without concurrency for 1000 requests:

	Rails	PHP	Flask	nodeJS
Time taken	227.283	66	236.008	384

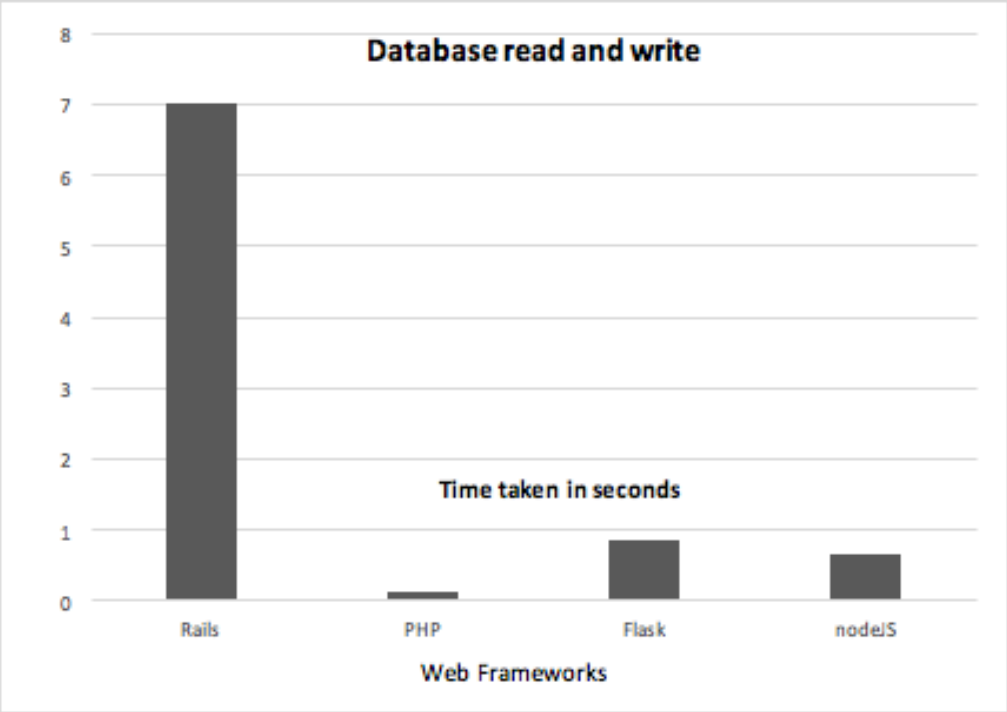
***time in seconds**



Database read and write

	Rails	PHP	Flask	nodeJS
Time taken	7	0.117	0.852	0.659

***time in seconds**



2. Security

Security Assessments:

1.**Cross Site Scripting (XSS)** - injecting javascript code in place of data might allow an attacker to execute code on the victim's browser, if the web server does not sanitize the input entered by the attacker. Some frameworks, like Rails perform HTML encoding on input data by default, in order to prevent XSS even when the programmer does not explicitly sanitize user input.

2.**Cross Site Request Forgery (CSRF)** - This is a very subtle attack that leverages a user's authenticated status on the website, to perform malicious actions. For example, a user authenticated into a bank, might receive a link that performs a transfer from the victim's account to the attacker's account. The browser also sends the authentication cookie along with this request since the domain refers to the same website. This results in the victim performing a transfer that they did not intend to perform. Once again some frameworks have built-in protections that include a pseudo-random token along with every form rendered by the server. This token is submitted along with the request and verified by the server to ensure that the

form itself was not crafted by an attacker. Predominantly, Php based web applications do not have these protections built-in and rely on the awareness of the programmer to prevent CSRF.

3. Cryptographic Security - This test attempts to identify security best practices involving password storage and session management. We observed the session tokens generated by the frameworks, to ensure that they were cryptographic random values. The passwords stored in the database were also observed to see whether they were hashed. This is a defense-in-depth solution that prevents password disclosure in the event that the database server gets breached.

4. Cookies - we inspected the cookies to identify properties that were set by the web application frameworks. This is also a defense-in-depth measure that prevents authentication cookies from being stolen in case the website has a vulnerability like cross-site scripting

A.Flask

1) Script Injections - xss - Attack Ineffective against application
2) Request Forgery - CSRF, OSRF - Susceptible, need to explicitly enforce security
3) Cryptographic Security - Sessions, passwords - Not present implicitly
4) Cookie Security - Setting it as HTTP only - Implicitly enforced

B. Rails

1) Script Injections - xss - Ineffective, Ruby encodes the data
2) Request Forgery - CSRF, OSRF - Uses anti-forgery tokens to protect against such attacks
3) Cryptographic Security - Sessions, passwords - Generates cryptographically random session tokens, for passwords, by default, it stores the hash in the database.
4) Cookie Security - Setting it as HTTP only - Implicitly enforced

C. PHP

1) Script Injections - xss - PHP unable to block such attacks.
2) Request Forgery - CSRF, OSRF - Php once again succumbed to this attack, beginning to see a pattern here.
3) Cryptographic Security - Sessions, passwords - No auto generation of hash, must be user-enforced.
4) Cookie Security - Setting it as HTTP only - Not set by default, must again be user-enforced.

D. NodeJS

1) Script Injections - xss - sanitizes input by default to prevent injections
2) Request Forgery - CSRF, OSRF - Susceptible by default, as it does not generate anti-forgery tokens

3) Cryptographic Security - Sessions, passwords - Does not hash passwords by default and requires the programmer to explicitly generate cryptographically secure sessions
4) Cookie Security - Setting it as HTTP only - Implicitly enforced

Flask - Siri and Aishwarya

Personal experience with Flask :

- Both developers began using Flask with only a limited knowledge of Python and absolutely no experience with Flask.
- We were pleasantly surprised to find that development with Flask proved smooth and quite simple. There are numerous resources available online which speeds up learning time.
- We observed that Flask is a very flexible framework and can be molded to suit a large variety of applications. While its underlying architecture is that of View-controller, it enforces no architecture or design decisions. Extensions and features are present in the form of third party libraries.
- Flask relieves the programmer of worrying about cookie security and attacks like cross-site injection. Flask uses a template engine, Jinja2 which automatically renders attacks like cross-site injection futile.
- Operations like file read/write which proved complicated and challenging in nodejs were extremely easy to implement using Flask.
- The only challenge we faced while developing in Flask was debugging. Flask implicitly does not print the stack trace in the event of an error and requires multiple try-except blocks and print debug statements to identify the source of an error.

Libraries and Package Management

Python's package manager *pip* provides an easy and convenient way to download packages and modules needed for extending the functionality of Flask. A large number of extensions are available, such as Flask-Admin, a simple and extensible administrative interface framework, Crypto.Cipher**,** for aes encryption of data, MongoKit module which provides an easy way to configure and connect to MongoD, to name a few.
Flask is well documented. On Github, they have 10,900 stars, and 5,000 questions tagged on StackOverflow.

Project Scalability

Adding a new functionality to a Flask application is relatively simple, and only involves adding a new route and controller.

Development Time

Familiarity with python is a huge advantage for using Flask. Flask allows exploiting the full

power of Python. In addition to this, the numerous packages, excellent documentation and large number of online tutorials for Flask application development makes Flask easy to master and use in a short period of time. Even someone unfamiliar with Python would be able to create an application reasonably fast with the help of these resources.

Ruby on Rails - Rohit and Kavya

Personal experience with Rails: (Rohit Gurunath)

- Rails as a framework is easy to learn and has extensive documentation. I was not a Ruby programmer before and had no idea about Rails until this project. But I was still able to learn the language well and implement a web application along with my teammates.
- The framework truly favors convention over configuration; something we were actually able to observe across various components of our application.
- There seems to be 'one prescribed' way of doing things in rails which is heavily advocated across the community. The upshot of following this convention is that it is very easy to understand code developed by other Rails programmers.
- Learning Ruby was very smooth and simple, which was probably because I had experience programming in python.
- The learning curve for rails was a bit steep initially until I got used to the 'Rails Way' of doing things. Once that was done, programming in rails was easy and elegant.
- There is an abundance of packages ('gems' as they are referred to in the rails community) that perform various functions like authentication, authorization, interfacing with multiple database systems like sqlite, mysql, mongodb etc. An interesting consequence of the 'convention over configuration' philosophy is that the various gems, follow best practices and are regularly updated with security patches. This has 2 advantages:
    - It relieves the programmer from having to worry about security issues arising from the Rails framework (not the application).
    - It ensures a good level of security for the application, without the programmers having to explicitly enable such features. Since the same best practices are followed across the community, simply installing the relevant 'gem' would take care of security end-to-end for the respective feature.
- This is corroborated by our findings with respect to security, whereby we observed that end-to-end security was taken care of through secure storage of passwords via bcrypt hashes, protection against request forgery and automatic html encoding of input data.
- The framework clearly protects against different , albeit not all, attacks. This ensures that an average rails application is quite secure compared to applications written in other languages, assuming the skill of the respective programmers is more or less the same.

- One pain point was that the encryption library's AES 256 cipher, which consistently threw a padding error. Even though the default padding mode for CBC mode encryption is PKCS7, it did not work as expected. However, once PKCS7 padding was implemented explicitly it worked as expected.

Personal experience with Rails: (Kavya Premkumar)

- Being a newbie to ruby and rails I found it relatively simple to learn the language and master the framework.
- The number of lines of code I had to write was extremely less compared to writing code in other languages/ frameworks. One advantage of using rails was that it generated a lot of code with very few commands.
- In addition to the standard models, views and controllers, rails also generates tests and database migrations that abstract the database being used. This allows us to change the database without worrying about conflicting schemas
- This clearly emphasizes software testing as a standard programming practice for rails applications. This also enforces programmers to perform tests on rails applications, which may be optional with other frameworks.
- It has good packages for working with databases like mongodb
- Also, for file upload, I was able to find a lot of packages(like paperClip, CarrierWave)to work with and tried to play around with them until I found the one that suited our app needs.
- I felt there was plenty of documentations and open source communities that provided help on topics that need clarification.
- Rails was found to be performing poorly in the benchmark tests. It had the worst server-side computation time for encrypting image files, which was indicative of its slow processing speed compared to other languages like php.

Libraries and Package Management

Rails uses 'gems', which are packages that include the modules, classes and version information. The versioning system for gems is very straightforward and requires a single command to install the appropriate gems. The framework manages packages and version control internally, which relieves the programmer from worrying about the same.

Scalability

Rails follows the MVC architecture with clean separation between models and controller actions. This allows the programmer to work with abstract database models, which makes the process of changing the database system simple. It provides various gems to work with the latest database packages like mongodb, and can be deployed easily on modern cloud service

providers. Thus the framework can easily handle a large project involving different heterogeneous components.

Development time

The time taken to develop the application was less as a lot of code was auto-generated by the framework. Various gems were available for authentication and working with mongodb, which clearly made the process of development easy. The same code can be used to work with different database systems, as rails internally handles generating database specific code to translate generic application code to database specific function calls. This relieves the programmer from having to learn different conventions for working with different database systems.

NodeJs - Siri and Somdeep

Personal experience with NodeJS

- We were not very familiar with Javascript when we started working on Nodejs. The set-up of Node and npm modules required for the web development took very little time. It was merely a command for every npm module that we had to install.
- Adding REST APIs in NodeJS was really simple because it follows model-view-controller architecture and Node works very well with MongoDb especially because of the JSON object support in javascript and the fact that MongoDb stores data as JSON objects.
- All the functionalities of HTTP GET/POST routes handling, HTTP request/response body parsing, crypto functions and middleware of our web application could be developed easily because the npm modules of the aforementioned functionalities are available and programmer-friendly.
- NodeJS performed very badly with respect to file I/O and we had to take extra care to ensure that we were doing the reads and writes synchronously because NodeJS does best for asynchronous operation.
- NodeJS has good documentation and tutorials available so code development and debugging was smooth.
- NodeJS also provides good debugging capabilities which helped us to find the server crashes and bugs easily. It provides stacktrace and informative errors which shortens the debugging time.

Libraries and Package Management

npm is the default package manager for Node. It is extremely powerful for the purpose of adding new dependencies and smoothly integrating this in the overall project flow. The general availabilities of libraries for Node is also extremely good given the strong open-source development community that exists for Node.

Project Scalability

Adding a new functionality to a NodeJs application is simple, and only involves adding a new route and the respective handling.

Development Time

The development time with NodeJS was more than that for Python which could be mainly attributed to the unfamiliarity of Javascript. Though numerous packages, excellent documentation and large number of online tutorials for NodeJS application development were available, it took us more time to understand the language intricacies and Node framework.

The asynchronous nature of Node that made it very different from all the other frameworks forced us to be more careful about our programming pathway for Node, for instance while writing the encryption functions , thereby increasing the associated development time. We got the feeling that the Node was geared to be architecturally expandable and efficient, more than in terms of pure performance or computation itself.

PHP - Somdeep Dey and Aishwarya Iyer

Personal Experience

Our personal experience working with PHP was a mixed bag and having limited previous knowledge of PHP, and as such we picked up a lot of information in the course of working on this.

An important distinction that we came to realize about PHP was that by itself, it is merely a scripting language that requires the added support of a complete web platform, for instance, Apache(which serves as the underlying server). We found the project organization in PHP extremely haphazard, the only way to actually enforce actual organization on it was by utilizing an overlying framework.

An interesting aspect of PHP's poor practices is an unexpected performance improvement it experiences. While carrying out server-side encryption, PHP was one of the best performing frameworks, as the results indicate. This gave us reason to wonder as to why it was so much more effective, until we realized that as PHP hosts precompiled binaries of all modules, execution or CPU-bound computation like the AES 128-bit encryption algorithm we ran was extremely effective. A strange case where an overhead for the developer turned into a fascinating runtime execution advantage for PHP.

Libraries and Package Management

One of the most difficult aspects of PHP development that we came across was the complexity of adding new packages or importing new libraries. For instance, in attempting to add the requisite drivers for connecting mongodb with PHP, it proved exceedingly complex as after downloading binaries, these had to be manually added to both configuration files and to the relevant php directories, making it extremely difficult for new users. The lack of any global

package or dependency manager (even with Composer, the process of installing composer itself was extremely complicated and poorly documented) greatly complicated environment setup.

Scalability

PHP programs worked well when smaller in scale(in terms of production code) but struggled as the scope of the application increases. A large reason for this is the code organisation followed by PHP, that involves a lack of modularity, exposes no modern concepts of HTTP routes or REST APIs, making it extremely difficult to design complex applications, necessitating a need to create new files for every route, an extremely complex procedure.

Development Time

The development time for general execution of tasks was good on PHP, as with dynamic typing, it becomes extremely easy to perform regular tasks. However, given the poor nature of integration with external libraries and the complicated procedure associated with dependency and package management, it became extremely difficult to limit the overall development time as environment setup consumed much of the total time spent on constructing the PHP based web app.

Things planned but not done

OAuth

Our original plan incorporated implementing OAuth 2.0 for each web framework . However in the early phase of implementation of this, we realized that using OAuth introduced an extra layer of security that would foil some of the attacks we planned. Considering that our main goal is to judge the competency of each framework, using OAuth would have been counterintuitive and would have added a layer of non-native security that is foreign to the default nature of such platforms, thus changing the nature of the tests we wanted to apply.

Video/Image Upload

While we did implement this feature, we decided not to benchmark it as we felt video rendering was the better test of the frameworks' performance. Part of this came from the mid-testing realization that for any upload, a POST request would be required but it would be unfeasible to replicate this in the form of a test script as filling the parameters for the POST request would be complicated. Also, the use of file download as opposed to upload seemed better as it gave us the entire idea of how efficiently the static files were served up by different frameworks as this was all we needed for the load testing.

Challenges faced

1. For nodejs, measuring accurate request service time was complicated by its asynchronous handling of requests. The Apache Testing Benchmark we used recorded the time a response code is received to be the request servicing time, which in nodejs does not correspond to the actual time taken for the request to be serviced. We had to use Google Chrome Developer tools to get an accurate measure of the actual servicing time.

2. Another challenge we faced was maintaining uniformity in the flow of request handling, method calls and file system accesses across the different frameworks to get as accurate a comparison as possible. Each framework has a different internal flow to servicing requests, for eg, in node js the most straightforward approach to encrypt an image involved a file read and write for every call. This naturally introduced an additional latency, rendering a fair comparison difficult. Thus we had to identify alternate strategies for performing encryption that most closely matched the number and type of computations as the other frameworks.

Conclusion

	Performance	Security	Libraries & Package Mgmt	Project Scalability	Development Time
node	**1 + * + 1 + ***	**2**	**2**	**3**	**1**
php	**4 + 4 + 3 + 3 (14)**	**0**	**2**	**0**	**0**
rails	**2 + 2 + 2 + 1 (7)**	**4**	**2**	**3**	**1**
flask	**3 + 1 + 4 + 2 (10)**	**2**	**2**	**3**	**1**

The matrix is laid out as follows:

- Higher is better
- For performance, each number represents the relative rank of that framework in the following categories
    - Number of requests
    - Concurrency
    - Server-side computation
    - Database access
- For security, we give a 0/1 for each of the following 4 categories
    - cross -site scripting
    - Cross site request forgery
    - Session handling and password storage

    - Cookie security
- We gave 0/1 for libraries and package management
- We gave 0/1 for project scalability, depending on
    - Integration with REST APIs
    - Deployment with modern cloud services
    - Availability of routing mechanisms to handle large projects and complex application logic

Overall Comparison :

Node:

- Advantages:
    - Good documentation, easy integration with cloud services, growing user-base
    - Good performance - only asynchronous though!
    - One Language to Rule them all - JavaScript
- Disadvantages:
    - Does not work well with large number of linear requests
    - Performance hit with synchronous operations - encryption

Rails:

- Advantages:
    - Offers best security - convention over configuration wins here!
    - Time to develop is less, availability of packages, support are high
    - Concurrency seems to be good - probably because of Apache
- Disadvantages:
    - Poor server-side performance - Interpreted language
    - DB interaction - poor
    - Does not scale

Php:

- Advantages:
    - Performance is better - Bytecode/ compiled modules
    - Documentation - good
- Disadvantages:
    - TIme to develop - very high
    - Learning curve is steep
    - Difficult to integrate with new database systems, cloud services
    - Extremely Insecure

Flask:

- Advantages:
    - Reasonable security - not as good as rails though!

- Time to develop is less; availability of packages, support are high
- Very efficient server-side computation

- Disadvantages:
  - Poor Concurrency - GIL
  - DB interaction - poor

Use Cases:

1) Rails : Twitter used rails initially to gain momentum. This helped them move to production very quickly at a time when they needed to ship many different features in a short span of time. However, they are now moving to scala in order to scale as rails does not scale very well.

2) Php : Facebook uses this extensively. However, they use hiphop to translate it to bytecode in order to improve performance. This doesn't seem to be a popular choice among startups owing to the lack of integration with modern REST APIs and cloud services.

3) Flask : Pinterest uses flask owing to the fewer number of lines needed and the lesser time to develop a full scale application. Flask also scales reasonably well and thus seems to be a popular choice among startups that favor ease of programming over extremely high performance.

4) nodeJS : Netflix uses node js extensively. Their rationale behind this was the easy integration with REST APIs, modern cloud and NoSQL database systems and the fact that programmers have to learn only one language - javascript. In addition, the microservice architecture suits node's design very well and allows them to scale efficiently.

References:

1] http://api.rubyonrails.org/ Website Title: Ruby on Rails API, Article Title: Ruby on Rails API

2] http://flask.pocoo.org/ Article Title: Flask, web development, one drop at a time, Date: 2010 - 2016, Author: Armin Ronacher

3] https://httpd.apache.org Website Title: Welcome!, Article Title: Essentials

4] http://php.net/docs.php Website Title: PHP, Article Title: Documentation

5] http://api.mongodb.org Website Title: MongoDB API, Article Title: MongoDB API

6]https://www.owasp.org/index.php/Cross-site_Scripting_(XSS) Website Title: OWASP, Article Title: Cross-site Scripting (XSS), Author: OWASP Foundation

7]https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF) Website Title: OWASP, Article Title: Cross-Site Request Forgery (CSRF), Author: OWASP Foundation

8] http://www.abhijainsblog.com/2015/04/over-posting-attack-in-mvc.html Website Title: Over Posting Attack in MVC, Article Title: Abhi Jain's .NET Blog, Date: Thursday, April 16, 2015, Author: Abhi Jain