Time Capsule Web Application

Team Members
Kevin Stewart
Patrick Mathieu
Quintin Leong

Project Motivation

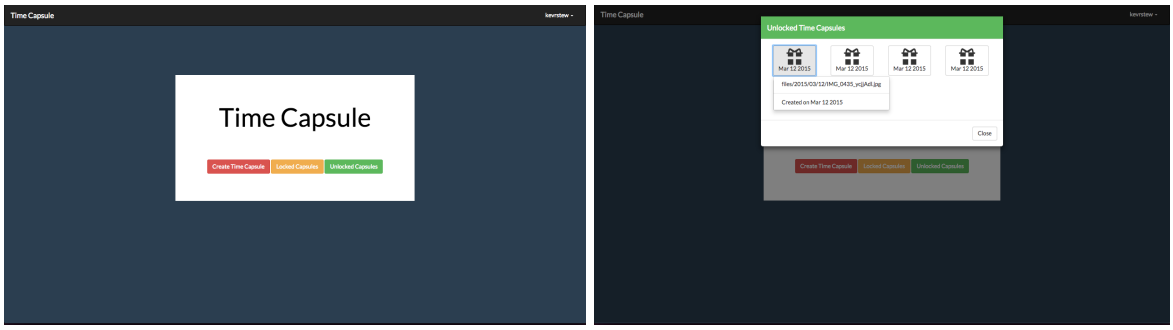
When our group first formed, we knew that we wanted to do some type of cross language comparison for our project. Additionally, since we are all graduating this year, we wanted to practice a skill that would somehow benefit us when we graduate. Since we were all back-end oriented computer science students that are planning to work for startups or larger tech companies, we decided to work on a building web applications. We hope that knowledge of how web applications are built will help us with future interviews and jobs. We wanted to make a web application using the technologies we hear so often when it comes to web development, so we decided that we would build an application using three different programming languages and web frameworks. One of the most talked about frameworks in web application development is Ruby on Rails, so we decided to make that our first language and framework of choice. For our second implementation, we decided to use PHP with the Codeigniter web framework because we wanted experience with the "LAMP stack" often discussed in the web application space. For our third and last implementation, we decided to use the Python language with the Django web framework. We chose Python and Django because of the rising popularity for developers with hands on Python/Django experience on job postings for web application engineer job searches.

Change of Plans

Originally we had planned to have Python, and Haskell be the two languages for our web application implementations, but we switched Haskell out for Php and decided to add Ruby as well to keep in line with the theme of web frameworks for interpreted languages. We felt that it was best because it would be harder to compare development times and difficulty of a compiled functional language to the other interpreted imperative approaches we would implement in terms of performance, development time, and learning curve. We also realized that we would run short on development time if we decided to to proceed with the Haskell approach because we underestimated the amount of work that would have to be put into the first implementation of Python.

Application Overview

Our web application, “Time Capsule”, is the digital version of a physical time capsule. A time capsule is often used to cache historic goods for a future generation to open and learn about the past. “Time Capsule” allows a user to attach media such as photos, videos, and documents of any sort and have our application save all the data. That data is then locked to the user until a date specified at the time of the capsule’s creation. Once the date specified is reached, the user is able to retrieve all of the data that they originally locked into their capsule and view or download their own personal “relics of the past”. Included in our web application are modules for facilitating user registration and login/logout actions, time capsule creation, viewing locked capsules with their unlock dates, and viewing/opening time capsules that have been unlocked.



In app screens taken from the Django implementation of the project

Major Features of Time Capsule

- User login system
- Interface to create a time capsule
- Interface to view locked time capsules and their unlock dates
- Interface to retrieve unlocked time capsules
- Data storage mechanism for capsule

System Architecture - MVC

Most web developers adopt a Model-view-controller (MVC) pattern for their web applications today. It creates a separation of concerns for the developer by splitting the code for the application into models, views, and controllers. The models contain all of the code that interacts with the database layer of the application. Views contain the embedded script code and HTML that is displayed to the client. Controllers are responsible for handling the logic and flow of the application. By separating these concerns, it makes for more structured, easy to understand code base speeding up the development and debugging process. When a client visits a web page, they are routed to a controller which then interacts with the required models and renders the view before sending the final result back to the client. This process repeats as the client is

routed to different controllers throughout the application. Each component does not retain it's state, so there is no memory of past transactions that have occurred inside of the models, views, or controllers.

Languages We Used

The first thing we noticed about the three technologies we chose was that they are all interpreted languages. So we did some research to find out why it seems like interpreted languages are preferable when building web applications. We noticed that *rapid development* was a common term used in the description for all three frameworks. So we tried to see if these were related. We found that the nature of interpreted languages contributed to the process of rapid development. A website that is backed by an interpreted language web framework does not require rebuilding, repackaging, and redeployment whenever there is a change made to the application's code. All three of those steps are required when working with a compiled language. One can simply just edit code and then swap out the necessary files on the machine hosting the application and have all of the changes happen immediately on the site. The downfall of interpreted languages is that they do not perform as well in terms of speed as compiled languages because their programs are not compiled into instructions of the target machine; interpreted languages are executed one statement at a time by another program that breaks the statements into subroutines that can then be executed in the target machine's code.

Ruby is a purely object oriented programming language. It was developed by Yukihiro Matsumoto and had its first release in 1994 as an alternative scripting language to Perl and Python. It is an interesting feature that every value in Ruby is an object, even primitive types. This means that every value belongs to a class that belongs to a superclass. At the top of the Ruby class hierarchy is the Object class like in the Java programming language. One difference from other object oriented programming languages is that objects do not have class methods. Classes have a set of methods that can be called on objects of that class instead. Object instances only have private variables that are only visible to itself as well, there is no notion of public instance fields in Ruby. This leads to all object communication taking place through method calls. Ruby is classified as an extremely dynamic programming language. Variables are able to hold any type of object inside of them. There is also a term called duck typing used to describe the behavior when Ruby is looking up a method call. Duck typing allows the programmer to make classes that imitate another class by implementing the same methods that are allowed to be used on the original class. This is possible because when Ruby is resolving which method to call, it does so by the name of the method without regard to the type of object it is being called on. This leads

to the notion that objects are not defined by what they are in Ruby, but the actions that can be performed on them. Hence the term duck typing that derives from the saying: “If it looks like a duck, swims like a duck, and quacks like a duck then it’s probably a duck.”

Php was created in 1994 by Ramsus Lerdorf as a server side scripting language designed specifically for web development. It has had wide popularity when developing websites and has been recorded to be present on over 240 million websites to date. Although not originally an object oriented programming language, object oriented features were added to the language in Php 3 released in the year 1998. Php is a dynamically, weak typed language where variables can hold any type of data in them and there is a predefined way in which variables of different types interact such as when trying to add the string “10” and the integer 5. Php will make sense of this operation using it’s predefined rules of interaction and will output the integer 15. The birth of what we know today as dynamic web pages was made possible by inserting Php code into static HTML files that would fill content dynamically on the server side before sending the result to the client’s browser.

Python is a general purpose programming language that was created in 1989 by Guido Van Rossum. It has a large emphasis on code readability with non-verbose syntax so things can be made with less lines of code. Python supports functional, object oriented, and imperative programming paradigms and is used in a variety of applications from client and server side programming to database programming. Python is a dynamically typed language where variables have no type and can hold any type of data. But despite being dynamic, it enforces strong typing unlike what we have just seen with Php. It would forbid the example above where the user wishes to add the string “10” and the integer 5 instead of trying to make sense of what the programmer meant by it. Python also uses duck typing like Ruby where objects are defined more by what they can do rather than what they are, but Python objects have class methods as well as strictly public instance fields unlike Ruby.

Technologies

The LAMP stack is an acronym that stands for Linux, Apache, MySQL, and Php. It is a popular choice in technologies used to develop websites because it is made up of all free and open-source software. This allows developers to use high performance software and duplicate the system for load balancing without having to pay any licensing fees that are required with proprietary software. Linux serves as the operating system. Apache is the web server that serves the web pages to clients accessing the server over the internet. MySQL is the database used to store persistent data for the web application. Php is the programming language used to interface with the MySQL

database and create dynamic content of the application. The LAMP stack is referred to as a “glued” framework because the developer is essentially combining multiple third party softwares together to create their own framework. All the pieces can be interchanged for other technologies. Nginx is a popular alternative for the Apache web server, MySQL can be replaced with any SQL database such as SQLite or PostgreSQL, and Php can be replaced with other scripting languages such as Perl or Python. Since all of the pieces that make up the software stack are so independent of each other, it was more difficult to get an initial development environment together due to configuration issues and lack of official documentation on how the technologies interact. After the initial time delay due to this reason, development went fairly smoothly. Codeigniter is a popular Php web framework with a traditional MVC design. Originally, we were using the Kohana framework because of the small footprint it has, hoping that the learning curve would not be too high. However, lack of good documentation led to us abandoning Kohana due to too much time being spent on scrounging the internet for solutions. Codeigniter, on the other hand, has a good amount of documentation and an active community which made the problem solving process easier even though the framework itself was more tedious to learn. The learning curve was fairly steep because we were also learning Php at the same time.

Ruby on Rails is a full stack framework that uses Ruby for back end processing and calculations. Having enough knowledge of the framework to reap the all of the benefits it has to offer can greatly simplify the development process, however, the learning curve for Rails is very steep. We found that in working with Rails there was more time spent figuring out how things worked rather than actually programming. This exhibits both the pros and cons of the framework. On the downside a newcomer will spend a good amount of time sifting Google search results in order to learn how to overcome basic challenges, but on the upside a seasoned user of Rails can develop an application at an amazing rate with very little code. Luckily, Rails is probably the most used full stack framework out there so there was plenty of documentation to refer to when needed. The Rails version of our application took about 30 hours to complete, but only required about 161 lines of code. 161 lines of code should only take about an hour or two to finish, which means that the rest of our time was spent figuring out how to provide functionality for user authentication, file uploading, and all other necessary system components. When rendering views, rails uses HTML files with embedded Ruby (.html.erb). This allowed for HTML to provide the basic structure of the page while Ruby provided dynamic information via embedded Ruby tags, which syntactically look like ‘<% Ruby code goes here %>’. Models in Rails can be created, altered and deleted via command-line generated migration files followed by a call to the command ‘rails

db:migrate'. Additional modification of database items can be performed within the Rails command shell without the need to write raw SQL.

Django is a framework for Python that offers a full stack approach to developing web applications. Architecturally, Django follows the MVC design pattern which is foundational to Django's capacity for rapid development. The reason we chose Django over other Python web frameworks is because of its extensive documentation, inclusion of systems that do the heavy lifting for common tasks in building a web application, and its straight forward templating system. Django's excellent documentation allowed us to get up and running with the framework almost immediately and more importantly offered us an excellent knowledge base to consult with when we were in need of help. As for some of its included systems, Django provides resources for facilitating most of the grunt work behind user registration and authentication, managing your database tables through an easy to use administrator portal, and for managing routes within your applications. This provided an excellent jump start to creating user models and controllers for accepting registration and login/logout requests. The easy to use ORM (object relational mapping) eliminated the need to write raw SQL at all. Creating and altering tables for our time capsules and assets was an incredibly smooth process due to the elegant migration system. Django's templating system was incredibly helpful for quickly tying our controllers into our views for easy and dynamic access to information stored in our database. Because of this, it allowed us to complete the project without the assistance of Ajax calls by just displaying python objects directly in our views. Within the templating system, Django makes this inclusion of Python amongst HTML, Javascript, and CSS, very simple. Two variants of bracketed python blocks are used, specifically the '{% statement %}' block tag which is used for control statements and the '{{ value }}' block tag for displaying values.

Development Process

We took an incremental approach when building our application. We built the application one framework at a time. For each framework, we first built a login and registration system that allowed clients to create a new user account and existing users to login in a secure manner. We then proceeded to build the minimum functioning logic of our application which allows users to create a new capsule with one item in it and store it in the database with a time to be unlocked. Then came the logic to fetch and display all capsules belonging to a user and allowing them to download capsules whose unlock time has been reached. Lastly, we implemented in the client side functionality to add additional capsule items through modifying the DOM (Document Object Model) tree to accommodate multiple items per capsule submission. The majority of this code

required client side Javascript to be achieved. Testing occurred between each one of these product phases to make sure the application was still doing what we intended it to do. Final product testing was done by black box testing each frameworks final implementation until proper functionality was achieved among all of the applications.

Comparing Development

Language	Development Time (Hours)	Lines of Code	Learning Curve
Ruby	~30	161	steep
Python	~20	259	moderate
Php	~25	~400	moderate

In terms of prior experience, we had none with the Ruby and Php or Rails and Codeigniter, and only some familiarity with Python and Django. Development across all three languages/frameworks varied in terms of development time, lines of code, and where efforts were most focused. Our experience in developing with Ruby on Rails was one that was mainly concerned with learning how it worked with intermittent sprints of work to build up the necessary systems. Due to the amount of heavy lifting that Rails does, it seems as though an application written in it is best served by a significant stage of planning before any programming implementation takes place in order to have more control over the applications inner workings.

With Django, a fairly even split of the work was struck between reading up on the documentation and actually building the application. Django seemed to hit a perfect balance in terms of the heavy lifting modern web frameworks provide. It was enough to get past the early stages of development quickly but not too much that it took away from our understanding of how certain systems were working. The trade off here is that slightly more code is necessary in Django than in Rails for similar tasks. However we feel this trade off is definitely worth it because of the time saved in reading documentation, which led to less development time overall, and the difference in code readability.

The Codeigniter framework was by far the most time intensive in terms of actual programming and ultimately led to the most lines of code out of all three implementations. However, the trade off here was that Php with Codeigniter offered the greatest amount of freedom and fine tuning in actually constructing our application. While Php as a whole seems to be a less than modern language for web development,

it is certainly a powerful language that has more than enough documentation to keep development from being derailed.

Below is a table charting out the main benefits and weaknesses we came across within each framework.

Ruby (on Rails)

Pros	Cons
Can get a lot done without the need of verbose syntax.	Excessive division of work between various controllers creates a bloated application structure.
Gemfile package management system is excellent.	Lots of the heavy lifting is done behind the scenes and abstracted in such a way that new Rails developers have a hard time understanding how everything ties together.

Python (Django)

Pros	Cons
Incredibly well documented language and framework, very active developer communities.	Relatively slow performance.
Elegant code that promotes readability and comprehension.	Django contains a lot of functionality that can in many cases bloat your application with unnecessary components.

Php (Codeigniter)

Pros	Cons
Very active community and great documentation.	Open source maintenance of language leads to inconsistent function declarations.
Associative arrays	Too many ways to accomplish a task leads to confusion over the best way to approach individual problems.

Conclusion

In conclusion, we learned a great deal about the structure of web applications and how they are built. Each language and framework has its benefits and downfalls. Ruby on Rails, although requiring the least amount of code, takes the most amount of time to build due to having to learn the complex framework. It takes much of the work out of the developer's hands with its under the hood magic and provides an extensive 'gem' library that leaves minimal implementation up to developers. Although, we can see the attraction for an experienced Rails developer as it leads to a very rapid development process. Php on the other hand allows for full control over every aspect of what happens in the application, but as a result the code is more verbose and takes nearly the same amount of time to develop as the Rails application. Python with Django is decidedly the best language and framework that hides boilerplate code and still exposes enough so that developers have a strong understanding of the what Django is doing behind the scenes. This leads to the fastest development time with an intermediate amount of lines of code compared to Ruby on Rails and Php. This project was very informative to us as engineers on the strengths and weaknesses of the most popular languages and web frameworks used to build web applications today.

Bibliography

http://en.wikipedia.org/wiki/PHP
http://en.wikibooks.org/wiki/Ruby_Programming/Overview
http://en.wikipedia.org/wiki/Interpreted_language
http://php.net/manual/en/history.php.php
http://python-history.blogspot.com/2009/01/introduction-and-overview.html
http://en.wikipedia.org/wiki/Python_%28programming_language%29#Syntax_and_semantics
https://en.wikipedia.org/wiki/Django_%28web_framework%29
https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller