

Master's Thesis

City College of New York

David Leonard

Date: May 2, 2016

Contents

1	Introduction	2
2	Version Control Systems	2
3	Benefits of VCS	2
3.1	Version Control	2
3.2	Issue Tracking	3
3.3	Best Practices	4
4	Project Types	4
4.1	Project Skeletons	4
4.2	Requirement Documents	4
5	Collaboration Evaluation	5
5.1	Metrics	5
5.2	Bitbucket API	5
5.3	Privacy and Authentication	6
6	Application Architecture	7
6.1	React.js	8
6.2	Node.js	9
6.2.1	Security Concerns	9
6.3	Express.js	10
6.4	MongoDB	11
6.4.1	Caching	11
6.5	Architecture Summary	11
7	Application Features	11
7.1	Authentication	11
7.2	Subscription Manager	11
7.3	Repository Visualization	12
7.3.1	Data Cards	13

1 Introduction

In the past year at the City College of New York, the Computer Science program has become immensely crowded - particularly the introductory computer science courses. Overcrowding of introductory classes leads to less feedback from Professors at a particularly vulnerable time for new students in which feedback is crucial for understanding the material. On the other hand, in the current generation of software engineering, learning how to use Version Control Systems (VCS) has become an absolute and necessary skill for all students studying computer science to learn. With these two points in mind, how can we leverage VCS to help alleviate the problem of providing feedback to students while at the same time teaching students how to successfully collaborate together? We will explore both of these ideas in this paper and provide a technical solution to address these problems.

2 Version Control Systems

In Software Engineering, we are faced with one common problem - how does a group of individuals successfully collaborate on a project while sharing one codebase? Using a VCS, we can achieve precisely that. Currently, there are two well-known VCS: ¹ git and ² mercurial. These tools allow developers to collaborate on a group of files in a **project** while solving problems such as **communication** and **merge conflicts** between files. These projects are stored as **repositories** in the **cloud** through the use of **collaborative platforms** which are ³ and ⁴. These platforms provide a service which allows developers to communicate, collaborate and control their codebase history. In this paper we will focus exclusively on using git repositories on Bitbucket.

3 Benefits of VCS

One common thing that students tend to do when working on software projects is to collaborate with other students. This allows them to learn from each other while at the same time having someone to bounce ideas off of in hopes of being able to solve their own problems. Since students already collaborate verbally with each other in class, one idea to handling overcrowding in classes would be to group students together. By having closely knit groups, students can learn from one another while at the same time being able to finish their numerous programming assignments.

As beneficial as VCS are, they are often neglected throughout the computer science curriculum. While they are often viewed as tools for software engineering in the real world, they have numerous benefits to students and professors alike. In the upcoming sections we explore the features of VCS and the Bitbucket platform that students can use to their advantage,

3.1 Version Control

The keywords in VCS are **version control**, which in this context means to have the ability to have **multiple versions** of an existing codebase. Suppose that a student is working on a programming assignment and continuously **commits** code into the repository, but at a certain point checks in code which breaks existing tests in the assignment. Through the power of *git*, the student can simply **roll back** their codebase to a previous commit at which everything was functioning properly.

The power of version control really shines in group projects, as students will regularly break existing code. Even if students commit code which breaks the entire codebase, it can be reverted back to a previous commit in which everything was working fine. Moreover, students can directly place comments at any line of code in the codebase if they have questions about how something works. By adopting this approach, students are able to learn more in-depth from one another as opposed to just grabbing code from elsewhere without properly dissecting it.

¹Git: <https://git-scm.com/>

²Mercurial: <https://www.mercurial-scm.org/>

³Github: <https://github.com/>

⁴Bitbucket: <https://bitbucket.org/>

```

technetium/technetium/bitbucket/unittests/test_bitmethods.py
Side-by-side diff View file Comment ...

48 48      """
49 49      Tests that URL has proper limit parameter
50 50      """
51 51 - match = 'https://bitbucket.org/api/1.0/repositories/technetiumccny/technetium/issues?limit=20&start=0'
51 51 + match = 'https://bitbucket.org/api/1.0/repositories/technetiumccny/technetium/issues?limit=20'
52 52 self.assertEqual(bitmethods.make_req_url
53 53     (self.user, self.repo, self.issues_endpt, limit=20), match)
54 54

56 56      """
57 57      Tests that URL has proper start parameter
58 58      """
59 59 - match = 'https://bitbucket.org/api/1.0/repositories/technetiumccny/technetium/issues?limit=50&start=20'
59 59 + match = 'https://bitbucket.org/api/1.0/repositories/technetiumccny/technetium/issues?limit=50'
60 60 self.assertEqual(bitmethods.make_req_url
61 61     (self.user, self.repo, self.issues_endpt, start=20), match)
61 61 + (self.user, self.repo, self.issues_endpt), match)
62 62
63 63 def test_make_req_url_with_limit_and_start(self):
64 64     """
65 65     Tests that URL is created with limit and start parameters
66 66     """
67 67 - match = 'https://bitbucket.org/api/1.0/repositories/technetiumccny/technetium/issues?limit=50&start=20'
67 67 + match = 'https://bitbucket.org/api/1.0/repositories/technetiumccny/technetium/issues?limit=50'
68 68 self.assertEqual(bitmethods.make_req_url
69 69     (self.user, self.repo, self.issues_endpt, limit=50, start=20), match)
69 69 + (self.user, self.repo, self.issues_endpt, limit=50), match)
70 70
71 71 def test_make_req_url_max_limit_50(self):
72 72     """

```

Figure 1: A sample commit

3.2 Issue Tracking

Bitbucket features a rich **issue tracker**, which allows individuals to create tickets detailing any bugs or features that need to be added to a project. The issue tracker can be utilized by creating a separate ticket for each feature that needs to be implemented in a given project:

Title	T	P	Status	Votes	Assignee	Created	Updated
#91: Linegraph cutoff dates	🔴	🟢	OPEN	1		2013-11-30	2014-05-03
#105: List index out of range when parsing data from git repo	🔴	🔴	NEW		David Leonard	2014-04-28	2014-05-03
#106: Fix for icons not on gravatar	🔴	🟢	RESOLVED		Jorge Yau	2014-04-28	2014-05-03
#107: Markdown support for issue tracker	🟢	🔴	NEW			2014-04-29	2014-04-29
#98: Implement Post new issue	🟢	🔴	WONTFIX		Jorge Yau	2013-12-04	2014-04-04
#103: Detailed issue with show more	🔴	🔴	RESOLVED		Jorge Yau	2013-12-10	2014-01-29
#82: Method to get Issue Comments in a Repository	🟢	🔴	DUPLICATE		Jorge Yau	2013-11-22	2013-12-12

Figure 2: A sample commit

As seen in Figure 2, additional labels may be added to an issue which show the status of the issue as well as the priority and assignee. By creating issues for all required features and distributing them amongst all group members, students can effectively split up a project in such a way that they can pick and choose which features they are comfortable with implementing. This approach helps to build the confidence in a student and ideally allows them to understand how other pieces of the program come together to form one coherent application. Additionally, issues may be closed directly by referencing them through commit messages:

This promotes clean and real-world software engineering practices, while teaching students how to effectively maintain a real codebase on a smaller scale. Another approach to using issue trackers is to cycle issues around to other group members should an individual gets stuck with a particular feature - this is to help




	Yeuk Hon Wong	dd41748	Fixed #124. Added pyramid_debugtoolbar back to setup.py	2014-08-07
	Yeuk Hon Wong	b07667d	Fixed #123. Make the sample configuration files more consistent.	2014-08-07
	Yeuk Hon Wong	bff34e	Fixed #105. Use uuid4().hex instead of our custom make_hex	2014-06-18

Figure 3: Resolving Issues directly through Commits

meet deadlines while also encouraging students to not give up and instead try their hand on another aspect of the program.

3.3 Best Practices

Students of all levels learn in their own personal way, and no two ways of thinking will always be the same. This is apparent when asking a student to implement a function which sums up all numbers in an array - two different approaches might be to use a *for* loop while another approach will use a *while* loop. In this case, neither approach is wrong or right, it is up to the student to decide. But what happens when students are given functions to implement which are more sophisticated? A professor will notice that students differ in:

- **Indentation:** Students may indent with 2 or 4 spaces, or in the *worst* case, left-justify everything.
- **Non-descriptive Variable Names:** Students may not use variable names which are descriptive, hence leading to confusion in the future.
- **Comments:** Students may not comment their code, which will cause problems in the future when reviewing for exams.

When working alone, students often do not gain insight as to how they may improve their skills or write readable code - a skill that has direct consequences when pursuing a career in software engineering. The best known way to remedy this is to read the source code of others, whether it be online, through peers or to read various programming books. In this model, students collaborating using VCS can observe how their peers write their own code and learn the best practices which work for them and overall get exposed to different ways of thinking.

4 Project Types

With the benefits of VCS covered, we now need to explore how students should appropriately create issue tickets for each feature which is needed in a given programming assignment. To do this, we explore various programming assignment formats.

4.1 Project Skeletons

A common format for giving students programming assignments is through the use of **project skeletons**, which consists of empty functions that are documented with comments regarding their implementations. In this scenario, it is trivial to create issues on Bitbucket for these assignments - every function which needs to be implemented will have a corresponding issue. To take things a step further, students can then assign these issues to themselves or other members in the group and add labels to functions which are blockers - meaning that their implementation is needed for other functions. In this sense, it allows students of all programming levels to effectively reason about their projects and to consider how to prioritize their implementation.

4.2 Requirement Documents

In higher level classes, sample code through project skeletons are often not given. In these situations, students are presented with documents which state the feature requirements that an application must satisfy. Despite

this, it is not always clear what path a project should take when started. Should a base class be built first, or helper functions? By having an issue tracker, students are encouraged to sit down and properly think about the project and break it down into smaller, more manageable chunks. Moreover, students will gain deeper insight as to why *feature x* should come before *feature y* and in turn will have an easier time with their assignment.

Overall, discussing projects at this granular level promotes a deeper understanding of what needs to be done. Often times students are thinking only about the end result and skip over the finer details, but this type of structured planning will save hours (if not days) of work. With the collaboration model fleshed out, we move onto our next problem - how can students properly be evaluated for any type of group project?

5 Collaboration Evaluation

The goal of collaboration is to have groups of students which effectively contribute to a group project. However, in a class full of students (and multiple sections), it becomes nearly impossible for a Professor to properly evaluate everyone fairly. In fact, it is hard enough to effectively determine if students have completed individual projects on their own. What about classes in which an individual's contributions to a group project determines their final grade? These are all difficult problems to solve in the classroom settings, however they become attainable through the use of version control. This brings us to the main focal point of the paper - implementing a tool which can effectively visualize contributions of individuals to a group project.

5.1 Metrics

As we have seen in Section 3, version control systems provide us with the ability to track issues and commits. However, we are not limited to these metrics alone - there are various other parameters which are readily made available to us:

- **Lines of Code:** How many lines of code did each student contribute to the project? This can be attributed to how much work has gone into their implementations.
- **Commit Comments:** How many comments on individual commits did each student create? These can be attributed to a student's curiosity and willingness to understand the codebase.
- **Issue Comments:** How many comments on feature issues did each student make? These are attributed to meaningful discussion of features, links to helpful resources and suggestions from other members.
- **Issues Closed:** How many issues were closed by each student? These correlate to how many features were implemented by each individual student.

Individually, these parameters are not particularly useful nor do they properly demonstrate the performance of a student. However, putting them all together will paint a meaningful and convincing story regarding the lifespan of a project as well as allowing Professors to understand how students are performing. We begin by describing the collection of these parameters for a given project.

5.2 Bitbucket API

Gathering the data needed for a given project is too tedious to be done by hand, however Bitbucket provides an **Application Program Interface (API)** which allows us to programmatically query data for a given project. An API is simply a web server which understands HTTP requests and responds with data from its database corresponding to the query parameters it has received. For example, the viewing the url <https://api.bitbucket.org/2.0/repositories/DrkSeph/wombat/commits/master?page=1> in any browser will respond with data corresponding to the commits in the repository *wombat* belonging to the user *DrkSeph*.

A subset of the output may be found below:

```
{
  values: [{
    hash: "bafefeeae9188400e144c88900db7196ae06165df",
    repository: {
      type: "repository",
      name: "wombat",
      full_name: "DrkSephy/wombat",
      uuid: "{2bb23825-18b6-4600-87b4-2f392536be76}"
    },
    author: {
      raw: "David Leonard <sephirothcloud1025@yahoo.com>",
      user: {
        username: "DrkSephy",
        display_name: "David Leonard",
        type: "user"
      }
    },
    date: "2016-04-07T21:50:34+00:00",
    message: "Merged in DrkSephy/fork-wombat (pull request #17) Adding h ",
    type: "commit"
  }
]}
```

Figure 4: Sample Data returned by Bitbucket API

Here, we can see various data regarding *commits* in the repository named *wombat*. Among this data, we can see every commit that has occurred along with the user who created the commit as well as the timestamp that it was created/updated. Using various other API endpoints, we can gather all of the necessary data to create an interactive visualization tool. However, we must discuss limitations of accessing data through the Bitbucket API.

5.3 Privacy and Authentication

A repository on Bitbucket may be set to *public* (any user can view the contents of the repository), or *private* in which nobody except the users given access to the repository may view its contents. Private repositories allow students to work in safety that no other individual can access their repository and view their code, preventing copying, cheating or other malicious intentions. However, accessing data from private repositories via the Bitbucket API requires a mechanism known as **authentication**, in which an application goes through a protocol to access protected resources. Bitbucket relies on **OAuth 2.0**, which is a protocol for successfully authenticating a client application with Bitbucket's server for accessing protected resources. The following steps are required for gaining access to these resources:

1. Create an OAuth consumer, which is our application in this case.
2. Using a given **secret key**, the consumer makes an HTTP request to get an authorization code.
3. Server (Bitbucket Server) issues an **Authorization Code**.
4. Consumer redirects the user to Bitbucket's login page using the Authorization code.
5. Resource owner (Bitbucket) grants access to the consumer (our application).
6. Server accepts / rejects the user authorization.
7. Consumer requests an **Access Token**.
8. Server responds with the **Access Token**.
9. Consumer can now access protected resources using this access token.
10. The server responds with the protected resource.

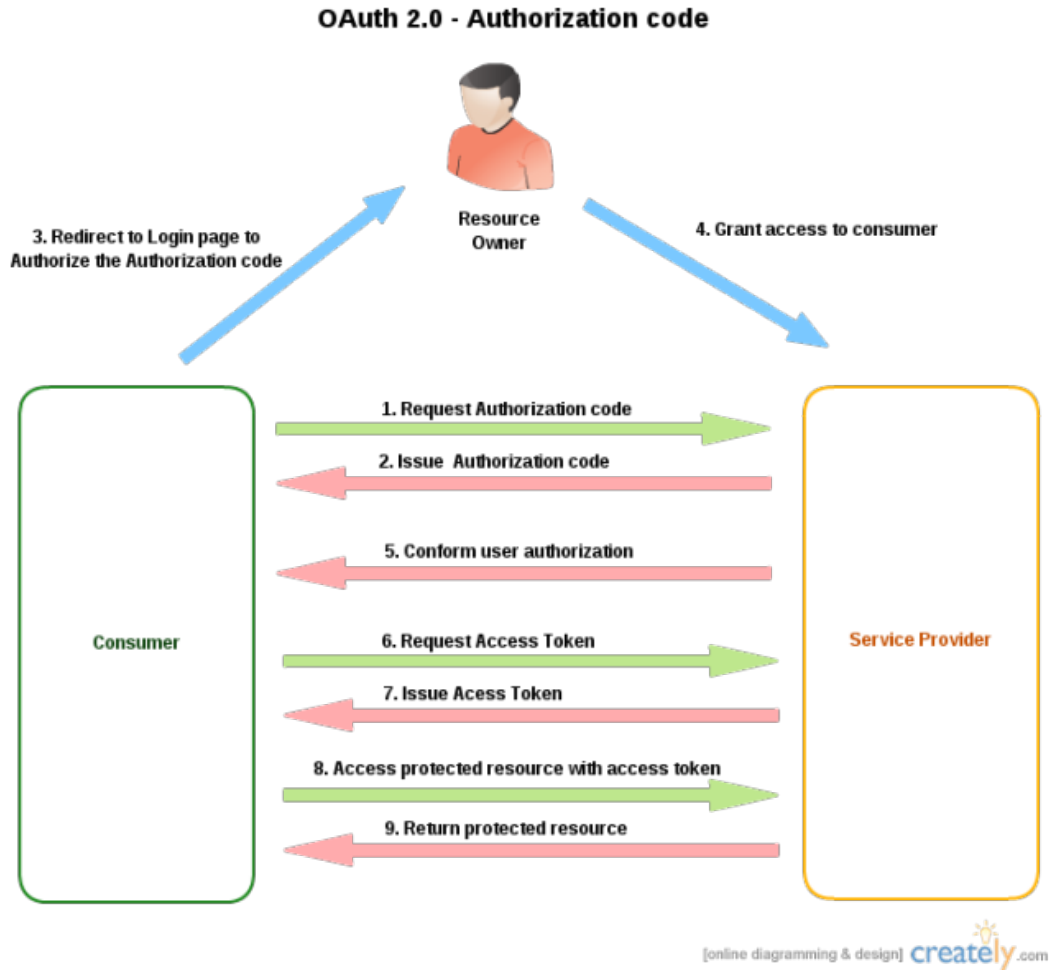


Figure 5: OAuth 2.0 Protocol

At the end of this entire protocol, steps 9 and 10 repeat until the access token expires. Once an access token expires, the consumer must refresh their token by re-negotiating with the server to gain a new access token.

6 Application Architecture

An important decision in our project is to properly choose the tools that are needed to handle the tasks required. In this section we explain the problems and decisions that went into deciding our architecture, which can be seen in Figure 6.



Figure 6: Application Architecture

The most important aspect of this project will be it's performance, both on the client-side and server-side. We begin by discussing performance of the client-side, with respect to rendering data to the client (web browser).

6.1 React.js

Given a class of 100 students with groups consisting of roughly 45 students, a Professor will have to visualize data for 2025 projects. As such, we will find ourselves rendering various data components which need to be efficient over time. Rendering data on the web browser requires modifying the **Document Object Model (DOM)** tree, which represents the layout of a web page.

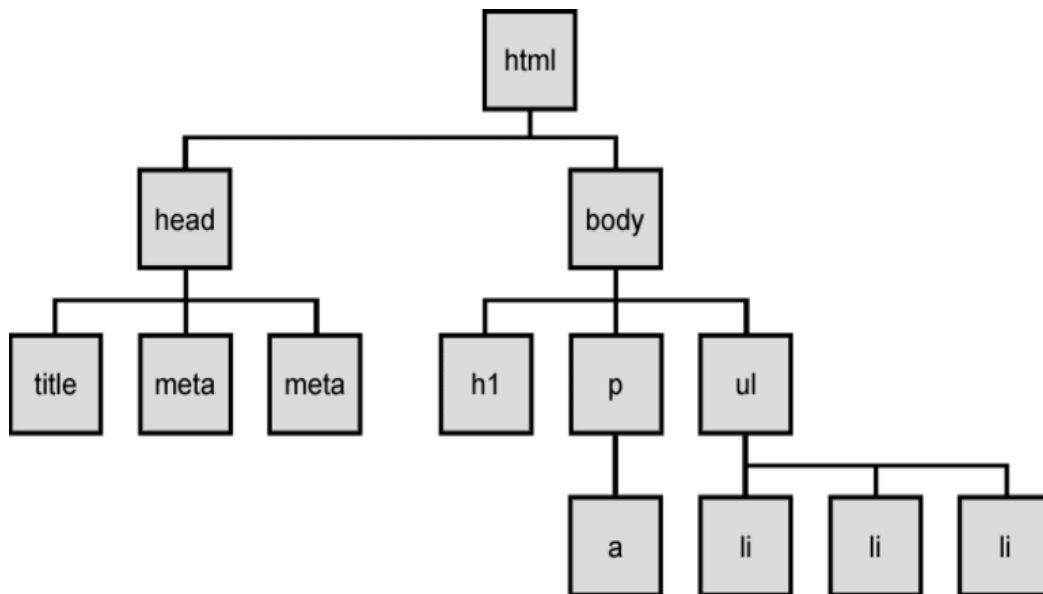


Figure 7: Document Object Model Tree

In terms of efficiency, updating the DOM tree consists of completely re-writing the tree from scratch. When updating the tree with new data and elements, this becomes costly in terms of node updates and rendering. Luckily, Facebook has released a cutting-edge library called **React** which is used for building sleek and efficient User Interfaces. React's power is it's ability to update the DOM tree in $\mathcal{O}(n)$ through it's efficient diffing algorithm. Instead of thrashing the entire DOM tree and rebuilding it from scratch, React computes a shadow DOM which shows the change in nodes from the actual DOM, and uses this to update the actual DOM by replacing/removing any nodes which have been marked for update. This problem, known as **reconciliation of the DOM** is asymptotically an $\mathcal{O}(n^3)$ problem that has been transformed to $\mathcal{O}(n)$ asymptotically.

At a high level, React's reconciliation works by using hand-crafted heuristics which help to predict which parts of the DOM tree are more likely to be updated. In our application, we will often find ourselves updating visualizations and datasets which will benefit from this efficient algorithm, which in turn will allow us to build a highly responsive user interface.

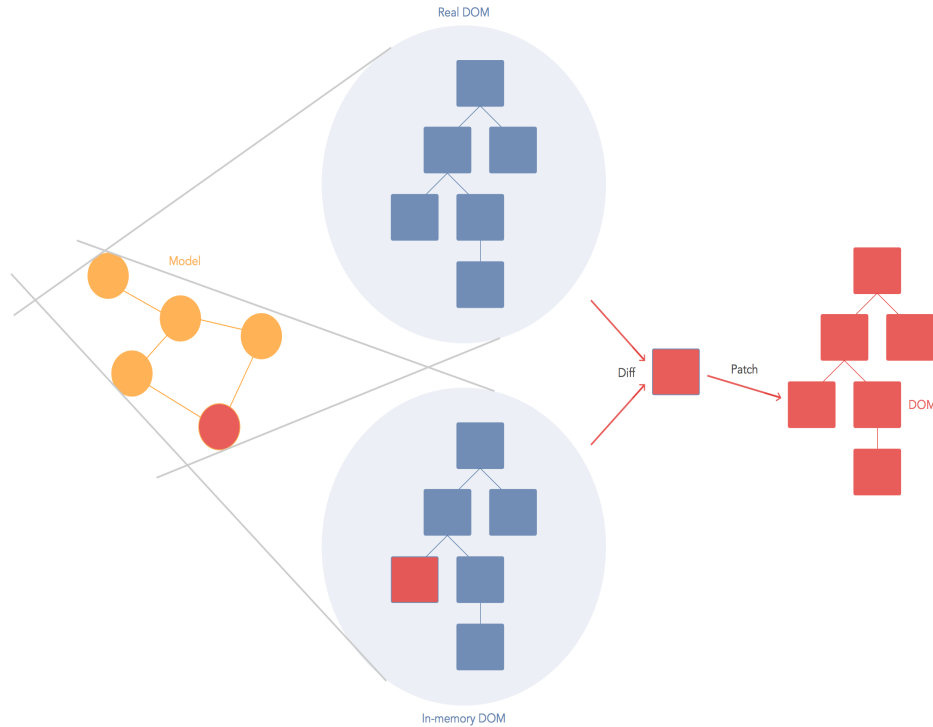


Figure 8: Reconciliation of the Document Object Model Tree

6.2 Node.js

Now that our client-side technology stack has been decided, we move to our server-side. The server will have the following tasks:

- **Authentication Flow:** The server will be needed for negotiating with Bitbucket to gain access tokens.
- **Re-negotiating Access Tokens:** When a user's access token has expired, the server will need to automatically re-negotiate for an access token.
- **Fetching Resources:** The server will be responsible for fetching all data through queries on the client-side application.
- **Communication with the database:** Whenever new data arrives, the server will communicate with the database in order to store responses.

The server also needs to be able to handle **concurrent requests** without blocking the main execution thread. Hence, we choose **Node** - an efficient and well-tested server-side implementation written in JavaScript. In fact, Node is optimized to use the V8 runtime, which is the fastest JavaScript rendering engine. With these features, we can asynchronously handle hundreds of requests to Bitbucket's server in order to fetch all of the data needed for our application in parallel.

6.2.1 Security Concerns

As we previously mentioned, a server is needed for handling Authentication Flow and re-negotiating access tokens. The reason for this is that it has been shown that pure client-side authentication flows are vulnerable

to various JavaScript attacks which can intercept and tamper with the request. Moreover, what does the client-side application do with the access token that is received after negotiating with the resource server? One solution is to store the token inside the browser through the use of a **cookie**, however this cookie is not secure and can be extracted using malicious JavaScript which in turn would lead to the application being compromised.

In order to solve this problem, using a server to handle our negotiation will lead to our application being secure from these malicious attacks. Lastly, we use the server to communicate with the database for storing user access tokens.

6.3 Express.js

In order to handle all of our API requests, we use **Express** which is an HTTP framework that is built on top of Node. This will allow us to send concurrent HTTP requests to the Bitbucket API which in turn will respond with the data we need to process. Express allows us to define simple HTTP routes, some of which are shown below:

```
/**
 * GET /api/count
 * Returns the number of commits in a repository.
 */
app.get('/api/count', isAuthenticated, (req, res) => {
  getJSON('resource_url', req.user.authToken)
    .then((data) => {
      res.send(data);
    });
});
```

```
/**
 * GET /api/issues
 * Returns issue data for a given repository.
 */
app.get('/api/issues', isAuthenticated, (req, res) => {
  getJSON('resource_url', req.user.authToken)
    .then((results) => {
      let parsedData = [];
      results['issues'].forEach((issue) => {
        let issueData = {};
        issueData.reported_by = issue.reported_by.username;
        issueData.title = issue.title;
        if (issue.responsible) {
          issueData.responsible = issue.responsible.username;
        }
        issueData.priority = issue.priority;
        issueData.metadata = issue.metadata.kind;
        issueData.id = generateRandomNumber();
        parsedData.push(issueData);
      });
      res.send(parsedData);
    });
});
```

6.4 MongoDB

The last component of our architecture is the database. In this application, we are primarily dealing with **JavaScript Object Notation (JSON)** - a lightweight response format returned from HTTP requests. Our data flow consists of the client-side application sending queries to the server, which in turn will respond with JSON data. This JSON is parsed and sent back to the client for rendering, and is also needs to be stored in the database for **caching** these web responses. JSON data consists of arbitrary key/value pairs, which makes it difficult to create schemas to store our data. As such, we use **MongoDB** - a document-based database which works very well with JSON data.

6.4.1 Caching

While the browser does a good job of caching repeated HTTP requests, we can speed up the turnaround from the client to the server and back by storing the parsed JSON on the server inside of the database. In this setup, whenever the client makes an HTTP request to the server, the server will first check if this data is already stored within the database - if not, the server queries the new data and updates the database and the client. Otherwise, the JSON is returned directly from the database and the client will update.

6.5 Architecture Summary

As we have described, we have an efficient pipeline ranging from client-side DOM rendering to concurrent HTTP requests on the server tied together seamlessly using MongoDB - all operating on a lightweight response format (JSON). In turn, we will be able to deliver a fast experience to the end-user which will consist of both Professors and Students.

7 Application Features

With our architecture decided, we move onto describing all of the features of our application. The end goal of this application is to give the viewer a high level understanding of the level of progress in a project. We also want students to be able to get immediate feedback as to how they are performing in a group project.

7.1 Authentication

When the user loads the application, they need to be able to authenticate with Bitbucket easily with one click. As stated, this is needed so that they can access any private repositories which they own. In order to implement the OAuth 2.0 protocol on the sever, we use a package called **passport.js**, which securely implements this protocol and allows us to fetch access tokens. Once the user is successfully authenticated with OAuth, they are redirected to the dashboard.

7.2 Subscription Manager

In order to generate visualizations for a project, the user needs a way to specify which repository they would like to process. We create a subscription manager component which displays all of the repositories which the authenticated user has read access to, shown in Figure 9.

Repositories	
Repository Name	Action
DrkSephy/client-certificate-authentication	<button>Subscribe</button>
DrkSephy/computer-organization	<button>Subscribe</button>
DrkSephy/data-structures-algorithms	<button>Subscribe</button>
DrkSephy/capstone-smart-house	<button>Subscribe</button>
DrkSephy/old-school-rpg-map	<button>Subscribe</button>

Figure 9: Subscription Manager

Here the user may quickly subscribe/unsubscribe to any of their repositories with the click of a button. While this is convenient for the user, it may be slow if they have many repositories to scroll through. To circumvent this, we have also created an alternative subscription manager shown in Figure 10 which allows the user to subscribe to a repository given a known repository name and Bitbucket username.

Add Subscription

Username

Repository Name

Figure 10: Alternate Subscription Manager

Once the user subscribes to a repository, the navigation bar will update with the list of subscribed repositories, shown in Figure 11.

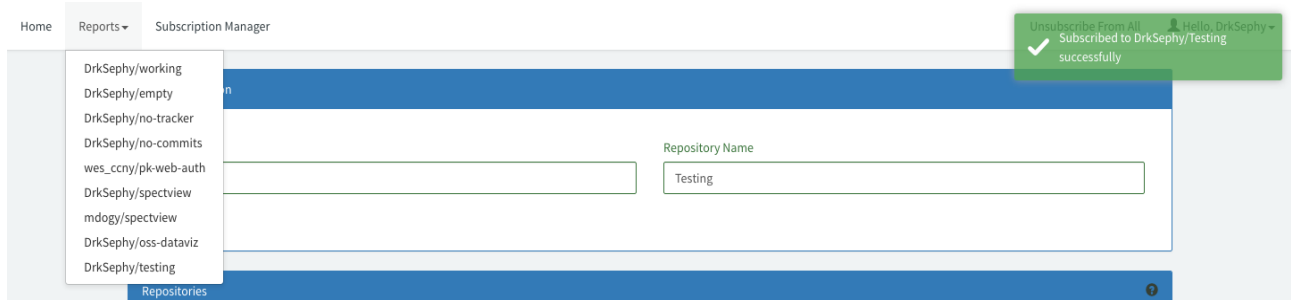


Figure 11: Navigation Bar Updates

Clicking one of the items in the navigation bar will take the user into the visualization of the clicked repository name, which we discuss next.

7.3 Repository Visualization

The repository visualization consists of the following data components:

- **Commits Card:** Displays the number of commits over a given time frame, along with a sparkling chart detailing the data trend.
- **Issues Opened:** Displays the number of issues opened over a given time frame, along with a sparkline chart detailing the data trend.
- **Issues Assigned:** Displays the number of issues assigned over a given time frame, along with a sparkline chart detailing the data trend.
- **Issues Closed:** Displays the number of issues closed over a given time frame, along with a sparkling chart detailing the data trend.
- **Repository Statistics:** Displays the totals of commits, issues opened/closed/assigned/comments and pull requests throughout the entire repository over the course of time.
- **Time Series:** Displays the commits over a one week for the given repository.

We break down their functionality over the course of the next sections.

7.3.1 Data Cards

The goal of the **commits**, **issues opened**, **issues closed** and the **issues assigned** cards is to give an overview of these datasets over a specified time.

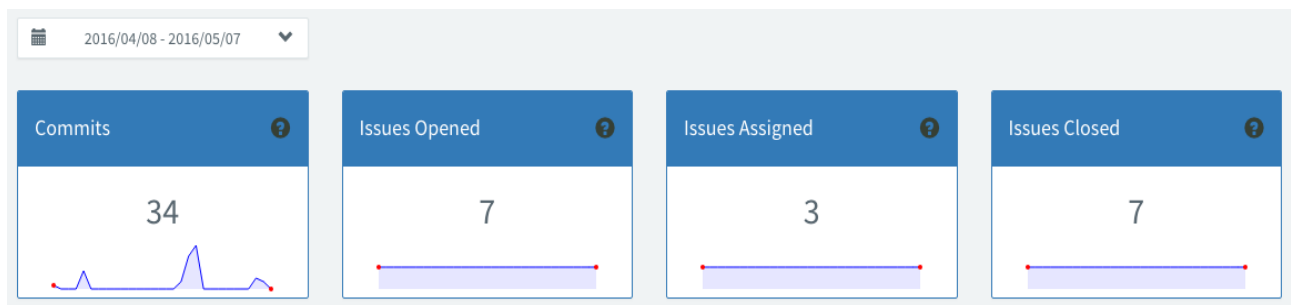


Figure 12: Data Cards

These parameters show arguably the most telling stories of the given repositories. Commits show that the repository is active, while the number of issues opened, assigned and closed demonstrate that features are being thought up and implemented throughout the given time frame. This data is further visualized through the underlying **sparkling charts**. Moreover, users can use the date range picker in the top left to specify a time range to

References

- [1] David Silver et. al. *Mastering the game of Go with deep neural networks and tree search*. (doi:10.1038/nature16961) January 28, 2016.
- [2] Campbell, M., Hoane, A, Hsu, F. *Deep Blue*. Artif. Intell. 134, 5783 (2002).
- [3] Google Research Blog *What we learned in Seoul with AlphaGo*. <https://goo.gl/bTT19m> March 16, 2016.