hammer-io

DESIGN DOCUMENT

Team: sdmay18-19
Client/Adviser: Lotfi Ben-Othmane
Members:
Erica Clark – Data Analytics Lead & Website/Content Management
Nathan De Graaf – Asana Expert & Weekly Status Reports
Nathan Karasch – Project Manager & Technical Writing
Jack Meyer – Communications & Software Architecture
Nischay Venkatram – UI Lead & Node.js SME
Email: sdmay18-19@iastate.edu
Website: sdmay18-19.sd.ece.iastate.edu

Revised: 5 Dec 2017 (version 2)

Contents

i. List of Tables and Figures

ii. Acronyms and Definitions

CI	Continuous Integration
CD	Continuous Deployment
CLI	Command Line Interface
DevOps	DevOps is a software engineering practice that aims at unifying software development (Dev) and software operation (Ops).[1]
Docker	A container platform used for deploying distributed software applications.
GUI	Graphical User Interface
MS	Microservice
Node.Js	A framework for running standalone JavaScript applications.
npm	Node Package Manager
PR	Pull Request
UI	User Interface

[1] https://en.wikipedia.org/wiki/DevOps

1 Introduction

1.1 Problem Statement

There is a tendency to develop software as a collection of managed microservices. The microservice architecture involves a fair amount of complexity that may intimidate small teams with limited resources, limited time, or limited domain knowledge. Often the microservices need to be deployed to the cloud. As a result, another constraint on a small team is maintaining the system as its various components are updated independently of each other. Traditionally this is managed by a separate "DevOps" team; however, again, a small team with limited resources may not be able to dedicate people to the task. These two primary concerns—the overhead involved in setting up a microservices architecture and the resources involved in maintaining one—outline the main problem this project is addressing: that a microservices architecture may not be feasible for small teams, such as students or startups.

1.2 Solution Approach

We will create a framework for developing Javascript-based microservices that exhibit specific quality attributes, as well as provide an automated DevOps process for managing the development and deployment of these JavaScript microservices. The project will be built in Node.js, an open-source server framework for running JavaScript code.

1.2.1 Purpose

By creating a framework for Node.js microservices, teams with limited knowledge, resources, and time can quickly get started with a simple infrastructure. We also provide the tools and structure to monitor and maintain the health of their applications in development and in production. This lets more developers start making cool things faster.

1.2.2 Goals

The main goal of our project is to create an environment to develop JavaScript microservices as well as a DevOps process to manage and monitor its deployment.

Specifically, we hope to create an app that requires minimal configuration and setup. This means the user would download one or two different tools and then have scripts create most of the configuration files needed with account info (e.g. GitHub, Docker Hub, etc.) supplied by the developers. Easy-to-use CLI/GUI tools would be available to push, deploy, and create new microservices.

Another goal that fits into this process is creating a seamless deployment workflow. This looks like a user pushing code to a master branch and watching his or her changes reflected in the live product a short while later. Once the application is deployed, the user then tracks the status of the application through a Data Monitoring app, which can notify interested parties when a service goes offline, display metrics related to server performance, etc. In addition, the Data Monitoring tool should be able to perform load balancing between microservices.

1.3 Operational Environment

Our project is a software project that will operate in 3 different environments:

1. A Command Line Interface (CLI), which will be installed on a user's computer
2. A user's linux server
3. A cloud-based hosting service

The CLI gives the user the ability to generate a Node.js microservices project. Since the CLI will be used to generate a new user project, it is necessary that it be installed on the user's computer. Having software which can be installed on a user's computer gives a semi-unpredictable operating environment. To make it more predictable and limit the scope of the project, our software supports running on the three most common operating systems (Windows 10, Mac OSX, and Debian-based distributions of Linux).

The second operating environment would be on a user's linux server. Our monitoring service will have the ability to be run on any hosting service a user should want to use, including their own servers. We must ensure that our monitoring service has the ability to be deployed on a user's server.

Finally, we provide a cloud-hosted solution for the online project generator and for the user project monitoring service. That means, in addition to the user's monitoring service being independently deployed to his own linux server, we will have a central monitoring service run on our own host.

Our software uses the Node.js programming language. Since we choose Node.js, this means our software will run in a JavaScript runtime environment built on Chrome's V8 JavaScript engine. Node.js requires the user of our software to have Node.js v5 or newer and npm installed on their computer/linux server.

Our project strives to be fully open-source, which makes distribution of the software slightly different than traditional software. We primarily use Node Package Manager (npm) to distribute our software, but all of the code is available from GitHub and can be built from source. The framework is also available via npm and can be installed from source. Finally, the monitoring service can also be distributed and installed from source.

1.4 Intended Users and Uses

For many companies, there is a lot of overhead in creating a new product. Our tool seeks to alleviate some of this overhead by providing a development pipeline out of the box. Our application and framework will allow developers to begin programming immediately without waiting to set up their deployment pipeline and have to worry about the intricacies of all the configurations. While many developers may benefit from Hammer, it is intended for students and small companies who need to deliver a working product with limited resources and time.

The framework will be highly customizable, with popular options like Express.js, GitHub, and TravisCI to fit a diverse set of needs. These options may be chosen initially or may be integrated after the project has already begun development.

1.5 Assumptions and Limitations

1.5.1 Assumptions

- We assumed the major audience that would be interested in using this product will be students, small teams, and startups that are looking for solutions that are open source, easy to use, and scalable. Most of the decisions that drive the project, are based primarily on this assumption. We want users to be able to scaffold projects quickly and scale without hitting any resource constraints. So all the technologies that we use in this project are open source and backed by a large community of developers.
- We assume the user is looking to build a microservice application in NodeJS, along with a continuous integration pipeline using TravisCI, containerization using Docker, backend using popular NodeJS web frameworks like ExpressJS, and deployment using hosting services like AWS, Heroku, and Digital Ocean. Hence, currently we only support generation of projects that use some or all of these technologies.
- If the user chooses to use any of these technologies, we assume they already have the necessary software installed to be able to generate and develop an application in these technologies. This includes having software like Git, NodeJS, etc installed and necessary accounts on Github, TravisCI, AWS, etc created beforehand.

1.5.2 Limitations

- The major limitation our project is that there is a limit to the processes we can automate. This can be seen directly from the assumptions. We cannot automate processes such as creating Github, TravisCI, or AWS accounts and also cannot simply install a variety of software on their machine. We merely assume they already have done that or fail gracefully.
- Since project revolves around bringing together open source software and many 3rd party services and API's, it is extremely difficult to test all the different pieces combined. This would make the application more prone to bugs.
- Since we are focusing on a certain set of technologies to provide to the user, we are possibly eliminating larger companies from the user group because they may want services or software options that we do not provide.

1.6 Expected End Product and Deliverables

By the end of development for the project, we will have created three products:

1. An automated DevOps process for NodeJS microservice applications
2. Application to monitor the health and status of the deployed NodeJS applications
3. A framework to develop NodeJS microservice applications

The first product we are developing is the automated DevOps process for NodeJS microservice applications. In this product, the user should have the ability to do the following:

- Manage their development workflow (Create Github repo, push code, etc)
- Have their code automatically pushed to a CI environment to have it tested and built
- Have their code automatically containerized and deployed to hosting services
- Manually deploy and revert deployments of their application
- Perform security analysis

1.6.1 Automated DevOps Process

The DevOps application can be used in two different ways. The first is as a command line interface. Here, the user will be able to have the initial code base for their project automatically generated (a.k.a. scaffolded). This should include the configuration files needed for the many services our application will be connecting to based on the user's choices. We will also hook up any 3rd party applications as needed.

The other way the DevOps capabilities are consumed is through a web application that allows users to view statistics about the DevOps process, view development artifacts (build statuses, code coverage, test results, test run history, etc.), manage the build and deployment of the various services, and perform the same functions as the CLI. The DevOps application can be deployed on the user's own server or be consumed through our cloud service.

1.6.2 Deployment Monitoring

The second part is the monitoring application. This overlaps with the Devops web application. We plan to have it as part of the same web application. The user should be able to view the health of all the deployed applications, have access to crash logs/reports, memory usage, CPU usage, application load, and uptime of each application.

1.6.3 NodeJS Microservice Development Framework

The third part is the framework that will allow users to write these microservice applications in NodeJS. The framework should have capabilities like service discovery, inter-service messaging, request load balancing, and service presence and health. This wil be researched further in the second semester. We are looking to leverage existing frameworks and modify it for our use case.

1.7 Deliverables Timeline

The following is a Gantt chart outlining the proposed timeline for the project's development through the first semester. The blue bars indicate project phases.

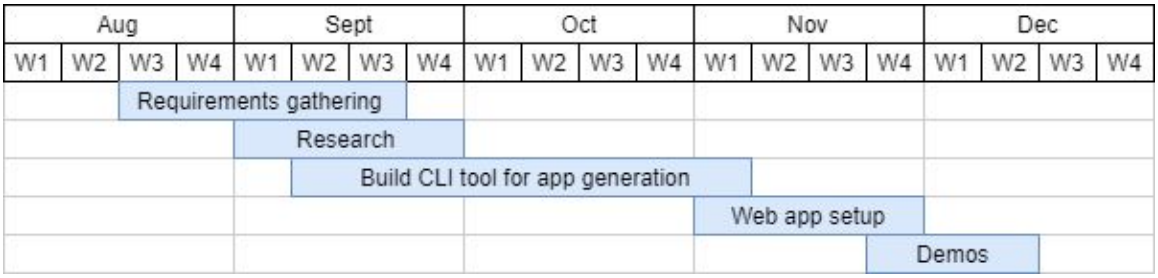

Figure 1. First Semester Timeline

The first few weeks of the project, we will spend time understanding the requirements, researching, and prototyping. Most of our design thinking will happen here and we'll make sure to meet with the client often in order to get the requirements. After we finish the first few weeks, we will start work on the command line tool to build an application and deploy it from scratch. We plan to spend a decent part of the semester building the CLI. Early November, we plan to transition to

architecting and building the web app where users can set up, deploy, and monitor the deployed applications. Most of the work for the web app will continue into the next semester.

The following is a Gantt chart outlining the proposed timeline for the project's development through the second semester. The blue bars indicate project phases.

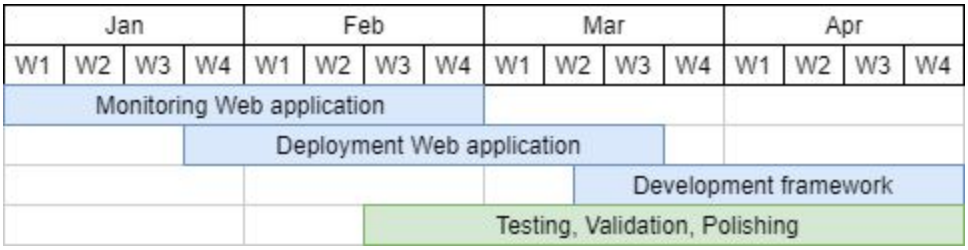

Figure 2. Second Semester Timeline

The plan for the first 2 months is to continue working on the web application to monitor and deploy microservice applications. There will be some overlap while working on the monitoring solution and deployment solution. We will continuously test and validate the solution. In early to mid March the plan is to slowly transition to developing the development framework that will allow users to code scalable microservice applications. We plan to wrap everything up by the end of April.

2 Specifications and Analysis

2.1 Functional Requirements

2.1.1 Automated DevOps process for Node.Js applications

- Provides a CLI, which contains commands to:
    - Setup a new project (generates default configuration files) based on the services that a user wants to include
    - Serve the web application version of the CLI, which must:
        - Allow the user to select which services he/she wants to include in a new project
        - Download the default files for a new project based on the services that a user wants to include
        - View reports, results, and statistics about the DevOps process
        - Manage the build and deployment of services
    - Automate delivery of code to the CI environment, which must:
        - Build the application
        - Package the application in a Dockerfile
        - Run test suites
        - Report status and results
- Ability to configure various services with a configuration file
- Provides documentation for configuration

2.1.2 Framework to develop Node.Js microservice applications

- Provides a CLI, which contains commands to:
    - Generate Node.Js templates for new microservices
    - Integrate new microservices into the existing microservice architecture
    - Configure the microservices (host, port, etc)

2.1.3 Monitoring Interface

- A web application, which must:
    - Allow reports and logs to be downloaded
    - Allows to view build statuses and test run history for different branches
    - Allows to view useful statistics like uptime, health, load, etc of the the deployed microservices
    - Allows to view time for completion of bug fixes or features.

2.2 Non-functional Requirements

- Usability
    - The system will only support the English language
    - The CLI must have a clean, consistent look and feel
    - The web application must have a clean, consistent look and feel
    - The application will be usable by those who have a limited understanding of DevOps
- Supportability
    - The system will support Unix-based systems (e.g. Mac or Linux)
    - The system will support Node.Js version 8.x
- Reliability
    - The deployed web applications will run 24 hours a day, 7 days a week
    - We expect to have an uptime of greater than 99%
- Security
    - The system will not store any plain-text passwords in configuration files or elsewhere

2.3 Standards

The team will use the following standards during development of the project:

- Version Control System
    - Git will be used as the primary means of version control for all project code.
    - The Google software suite (Docs, Sheets, etc) will be used for all formatted documentation, such as planning and design documents. It has version control features that can accessed from the menu (File → Version history).
- Code Review
    - Development will be done in feature branches.
    - When a feature branch is ready to be merged to the master branch, the author will assign one or more of the other project members as a reviewer.
    - Reviewers must check to ensure the code:
        - correctly implements the desired functionality

        - contains sufficient tests to ensure correctness
        - passes tests
        - is free of errors
        - integrates successfully with the existing software
    - Reviewers will communicate code issues to the author, who is then responsible for addressing all issues.
    - Once the code has passed inspection, it can be merged to master.
- Testing Standards
    - Mocha for Unit Testing, Acceptance Testing, and Integration Testing
        - It is expected that all code is tested to the best of the developer's abilities
    - Manual Testing for System Acceptance Testing
        - Weekly project demos will happen at meetings with client to determine if the new features are what was expected
    - Manual Testing to fill any gaps in automation testing

2.4 Proposed Design

The team has finished the minimal viable product of the automated DevOps CLI (the first major Functional Requirement). The following has been accomplished thus far:

- When the user starts the CLI, it asks a series of questions for scaffolding a new project. It asks for the following:
    - Project Name, Description, Version, Author, and License
    - Tooling Options
        - Choice of Continuous Integration Tool
            - Currently we support TravisCI
        - Choice of Containerization Tool
            - Currently we support Docker
        - Choice of Hosting Service
            - Currently we support Heroku
        - Choice of Web Application Framework
            - Currently we support ExpressJS
    - GitHub username/password
    - For each tooling option selected above, the CLI asks follow up questions based on the user's choice (e.g. if Heroku is chosen for hosting, it will prompt for the user's Heroku email, username, and password)
- Once the prompts are complete, the CLI generates the initial file structure:
    - It creates a new directory for the user's project
    - It generates the "package.json" and "src/index.js" files for the NodeJS project
    - It generates a ".gitignore" file
    - If ExpressJS is chosen as the web application framework, the CLI will add Express to the module dependencies and generate the initial "src/routes.js" file
    - If TravisCI was chosen, the CLI will generate a ".travis.yml" file for the project
    - If Docker was chosen, the CLI will generate a "Dockerfile" and ".dockerignore"

- Once the files have been initialized, the CLI integrates the project with the selected third-party applications
    - It initializes a new git repo in the project folder, commits the files, and pushes them to a new repository in the user's account
    - It enables the project in the user's TravisCI account
    - It creates a new Heroku project
    - It builds the project on TravisCI
    - It deploys application to Heroku on success build

The team has successfully deployed the CLI to NPM and can be downloaded using the npm install command via the command line. Currently, the support for the CLI is ongoing and the team is adding features when requested and fixing bugs when found. Below you can find a flowchart for the flow of the program from a user's point of view.

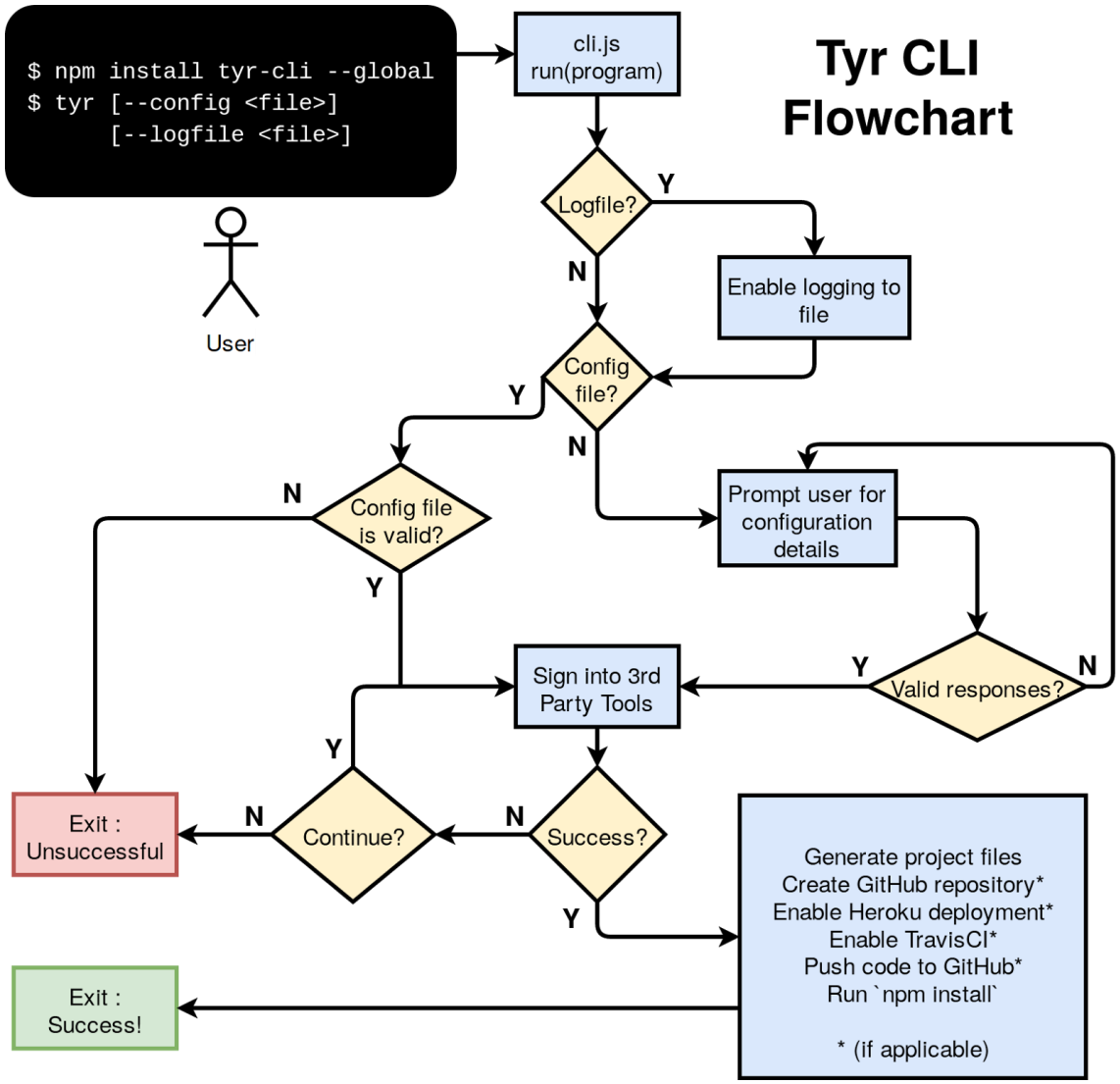

Figure 3: Tyr CLI Flowchart

The following is the proposed system block diagram for the deployed web application portion of the project. The user has the option of generating a project from the CLI or downloading a zip archive of the files generated by the web platform. Once the project is created, it gets pushed to Source Control (GitHub) by the CLI, enabled in the Continuous Integration suite (TravisCI), and deployed to the Cloud Hosting platform (Heroku). The TravisCI automation, which runs static and dynamic tests, is triggered by web hooks in the GitHub repository. Tests are run on code pushes, merges, and pull requests, and upon merging to master it is setup to deploy a docker container of the app to Heroku.

The DevOps Monitoring Platform provides the graphical user interface to monitor various aspects of the project: hosting uptime, test results, code coverage, repository statistics, etc.

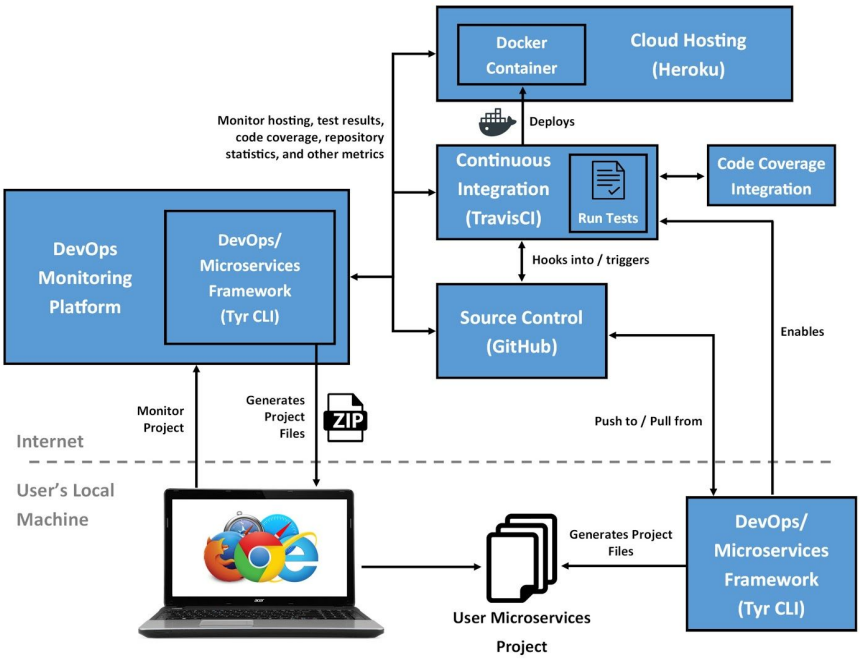

Figure 4: System Block Diagram

The team is currently working on the next phase of the project, which includes developing a web application to do the following:

- The ability to create users, login, and logout
- Duplicate the functionality of the CLI
- The ability to to create, edit, and remove
- The ability to view all projects
- The ability to add and remove contributors on a project
- View statistics about the project such as build statuses, test results, code coverage, issue tracking information, and more.

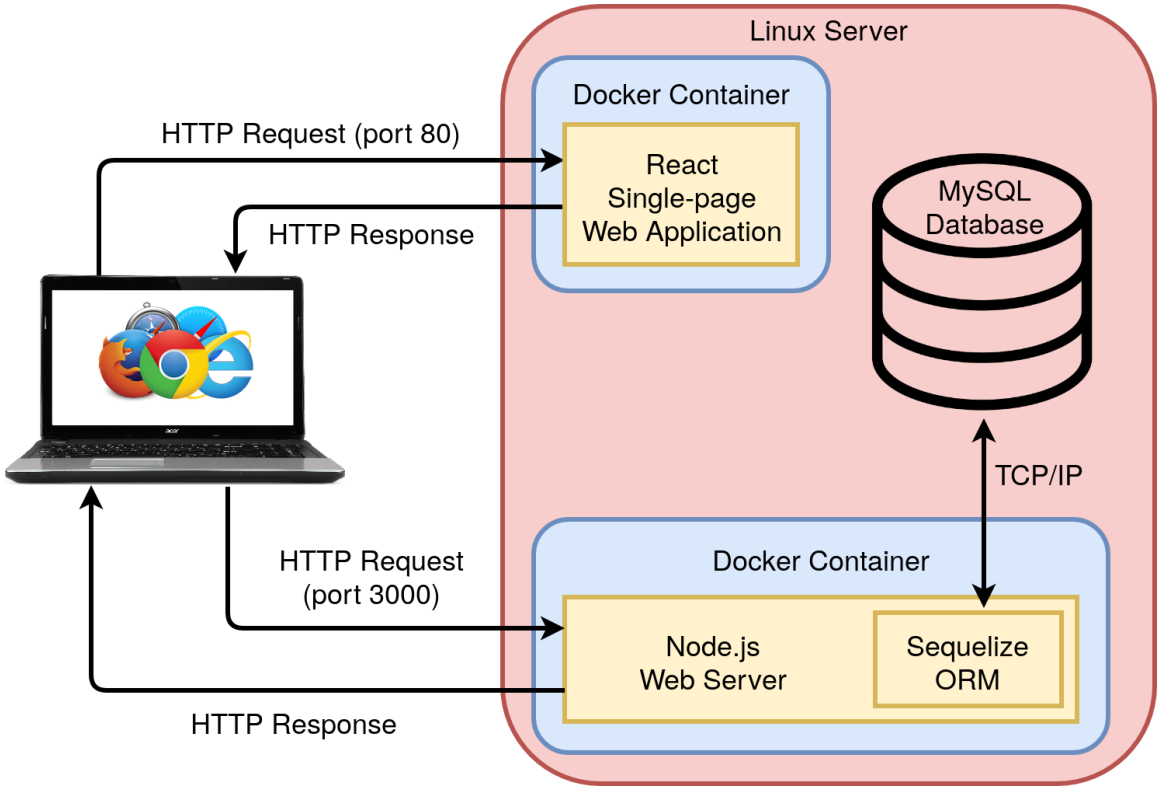

Figure 5: Software Architecture of Yggdrasil (discussed below)

2.5 Design Analysis

Our team has created the minimum viable framework for the automation of the creation and deployment of a simple Node.js project. The work done so far mainly consists of linking various services together through a simple CLI for the user. Targeted research and careful deliberation was used to pick the different tools our framework would use.

The simple CLI we have created has functionality for creating a sample project along with all the necessary configuration files for the selected services. So far, we provide TravisCI as an option for continuous integration and Docker as an option for a containerization platform. GitHub will act as the software development platform for the user's project. After the user provides their account details, our CLI sets up both the user's local git structure along with a connected remote repository on their account. This GitHub project and repository is then linked with TravisCI and in turn with Docker. This overall allows the user to go from local development to their remote project all the way to a container within Docker. The CLI has been able to successfully perform each step of this functionality.

Our team makes the conscious choice towards opinionated development. This means, for example, that rather than work to support all forms of continuous integration services, we choose to support TravisCI in the best way possible and leave the option for other tools to be added in the future. This practice of choosing the simplest or most popular tool used in the industry best fits the needs of our client who wants the minimum hassle possible in setting up and using the services of their

project. This means that at each part in this process we have the option to continue to add options and services for our users.

Our team is also working on the web application. Most of the tools that our framework will support were chosen when we developed the CLI. Part of the web application's functionality is to duplicate what the CLI can do. The web application is going to be developed using JavaScript, with the React framework used on the client side and Node.js used on the server side, our database is a MySQL database. We choose JavaScript because the team is all familiar with the language and our tool will support the creation of a JavaScript project. MySQL is used because of its ease of use and will store the data in a easy to use format. Our project will be wrapped with a Docker Container which will make it easy to deploy in many types of environments. The project will use TravisCI for its Continuous Integration and Continuous Deployment. The TravisCI plus Docker setup will allow us, as developers, to focus only on development after the tools are set up. This will allow for quicker development and eliminate production bugs. We anticipate the both the front end server and back end server will be deployed on a Linux server.

As we move forward, we will continue to build out the services and functionality for DevOps and deployment of the JavaScript services. We will continue to build out the monitoring and DevOps functionality through a more extensive front end website to supplement the simple CLI we have built so far.

3 Testing and Implementation

We will perform several types of testing to ensure the project meets the functional and nonfunctional requirements. As needed, we will reiterate on the development and design of new features if they fail any of the various tests detailed below.

- Regression Tests:
  - Unit Tests:
    - Unit tests shall be written for all code committed to the repository.
    - The author of the code is responsible for its unit tests, and the code reviewers are responsible for holding the author accountable.
  - Integration Tests:
    - Once a significant number of components have been integrated, tasks will be created and assigned to developers to write integration tests.
    - The integration tests should ensure the subset of components behave together as intended.
- System and Acceptance Testing:
  - Manual testing by developers will be the standard for System Testing. This will take place upon merging new code and upon completion of major portions of functionality.
  - Product demos to the client will be the standard for Acceptance Testing. These take place on a weekly basis at the team meeting.
- Code Coverage:
  - We will install a code coverage tool into the continuous integration suite to report what level of test coverage we have achieved. Given the short time frame of the project and the rapid pace at which a new project is developed, we likely won't shoot for a hard code coverage metric until the second semester. At that point we will evaluate whether code coverage makes sense in the context of our project and what percentage we should be targeting.

One of the testing challenges we face is that we rely on tight integration with third-party distributed applications for our software to function. For example, the CLI creates a new GitHub repository through GitHub's REST API and enables TravisCI on that repository using TravisCI's REST API. It doesn't stop there. We will need to integrate with hosting services (e.g. Heroku), containerization services (e.g. DockerHub) and other services that are part of the DevOps pipeline. These integrations are vital to meeting our functional requirements, but they are very difficult to automate. We created tests that run on TravisCI when built, however, occasionally these tests will fail if we make too many requests to GitHub so TravisCI does not fail the entire build if these integration tests do not pass. However, they ensure that the developer does not make breaking changes to one part of the system while altering the deployment pipeline.

3.1 Interface Specifications

As we develop our project, we add tests to verify consistent functionality after each change. A majority of our tests code interfaces with various API including GitHub, Heroku, and TravisCI. To ensure continued functionality, we have a suite of integration tests that test that the calls to the API perform correctly and bring about the expected result.

3.2 Hardware and software

Mocha: A test suite for Node.js that results in organized, and easy-to-read tests. The tests are run each time there is a pull request to master to ensure that the master branch always has functional code.

Linter: Enables consistent style between multiple developers and keeps track if there are any remaining unused imports, etc to keep the code base clean.

CodeCov: Analyzes and reports our test coverage. It alerts us if any changes were made that reduce the percentage of our code that is tested to improve the quality of our tests and ensure that all parts of the code is being tested.

TravicCI: The continuous integration tool that we set up a new project to use. After making requests to the API during testing, we make more requests to ensure that the changes were made. Additionally, for our own code, Travis CI will run the builds to ensure that the code on GitHub remains in a functional state by running our suite of tests.

GitHub: The user's project files will be pushed to GitHub. We will need to verify that the project files exist on GitHub.

Heroku: This is the server where the user's project will be deployed. Our testing must verify that a project has been created on Heroku and the correct settings are enabled.

3.3 Process

The testing starts locally on the developer's machine. First, it is only unit tests to ensure that the new code functions as expected. Before it is pushed, Regression Testing is done using Unit and Integration testing to ensure that the functionality of other parts of the code base have not changed unexpectedly. If these tests pass, the code is pushed to GitHub, where the tests are ran again to ensure that still no changes were unexpectedly made while merging and to ensure that the code on GitHub is functional. If the build fails on Travis, the code will not be merged into the master branch on GitHub until it works as expected. Once the corrected code has been committed and the tests pass in TravisCI, the PR can be merged after it has been reviewed by another developer.

Once a project has passed the Travis build, it can be deployed in the case of the webapp, or published to npm in the case of the command line interface. It then must undergo System Acceptance Testing. The developers then present the new feature or developments to the client who can decide whether or not it fulfills the required functionality. If it does, the developers may move on to a new solution. Otherwise, the developers will need to tweak or redesign and implement the current solution according to the client's feedback.

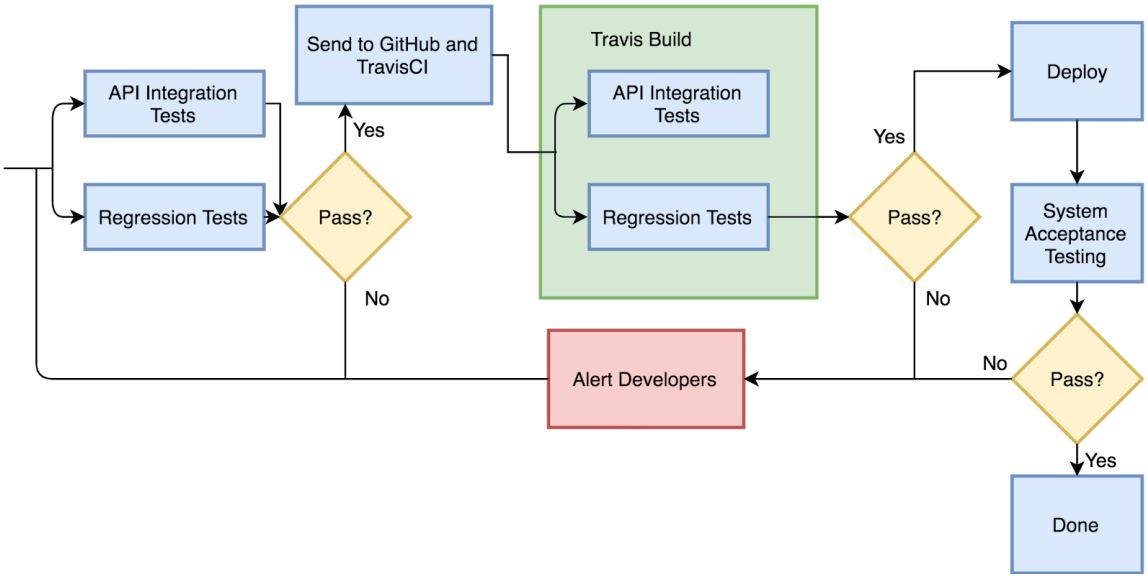

Figure 6. Test Flow Diagram (discussed below)

3.4 Testing Flow

1. Regression testing and API Integration Tests
    a. If the unit tests pass and integration tests pass, we push the project to GitHub and TravisCI begins to build it.
    b. Otherwise, we rerun the tests
2. Travis Build - API Integration Testing and Regression tests
    a. If they pass successfully, the project is deployed (for the webapp).
    b. If it fails, an email is sent out to the developers, and they have to start the process over to fix the failure.
3. System Acceptance Testing
    a. If it does pass, the process is over for that feature.
    b. If it does not pass the client's approval, any deployments may need to be rolled back and the developers must redevelop the solution.

3.5 Results

We have established a rudimentary test cycle centered around unit testing, style analysis testing, integration testing, code reviews, manual testing and system and acceptance testing. The team is adhering well to this initial process, and we are currently researching additional test methods for the various third-party integrations.

The build and test logs from TravisCI can be viewed at https://travis-ci.org/hammer-io/tyr .

[This part will be refined in the 492 semester where the majority of the implementation and testing work will take place]

4 Closing Material

4.1 Conclusion

The popularity, simplicity, and usefulness of JavaScript microservices means there is a need from users for an easy to use and an all encompassing framework and devops systems to build and deploy these JavaScript services. The goal of this project is to meet these needs. Hammer seeks to provide three main parts: an automated deployment process, a Node.js framework to accommodate and enable microservices, and a data monitoring platform to monitor the health and metrics of the entire system.

In order to achieve this functionality, our team plans to implement the following three parts of the overall product. First we build a DevOps application to manage the code base of the service. This can be managed through either a CLI or a web application. Building these applications will be the first step of the project. Once finished with this portion, the user should be able to deploy their own service to be consumed by their users through our application. Second, we will create a monitoring application that will build upon the DevOps web application. This shows the users all relevant information about their information including logs, load, uptime, etc. This monitoring application is part of the features that makes our application better for our clients than managing each of these services individually. Finally, we create a framework to allow users to write these microservices in NodeJS. The framework details will be further developed during the second semester.

Hammer, our solution to all JavaScript microservice development needs, seeks to meet the specific needs of our users in a way that other similar services cannot match. Our application's opinionated approach of setting up a suggested approach without the user needing any specialized knowledge makes Hammer the perfect choice for students or small teams that cannot afford to waste time or resources setting up and managing the logistics of their application. By focusing on simplicity, ease of use, and specializing in JavaScript microservices, our application can provide a better service than any other general DevOps or development framework service.

4.2 References

- Babel Documentation: https://babeljs.io/docs/usage/babelrc/
- Docker Documentation: https://docs.docker.com/develop/sdk/
- DockerHub API: https://docs.docker.com/v1.4/reference/api/docker-io_api/
- ESLint Documentation: https://eslint.org/docs/user-guide/
- ExpressJS Documentation: https://expressjs.com/en/4x/api.html
- GitHub API: https://developer.github.com/v3/
- Heroku API: https://devcenter.heroku.com/categories/platform-api
- Mocha Documentation: https://mochajs.org/
- NodeJS Documentation: https://nodejs.org/en/docs/
- NPM Documentation: https://docs.npmjs.com/
- TravisCI API: https://docs.travis-ci.com/api
- TravisCI Documentation: https://docs.travis-ci.com/