Hammer-IO Final Report

23 April 2018

Team Number	19
Team Email	sdmay18-19@iastate.edu
Team Website	sdmay18-19.sd.ece.iastae.edu (hammer-io.github.io)
Client/Adviser	Dr. Lotfi ben-Othmane

Developers

Erica Clark

Nathan De Graaf

Nathan Karasch

Jack Meyer

Nischay Venkatram

Table of Contents

Table of Figures

Introduction

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.

Goal

The main goal of our project is to give a user or team with limited resources the ability to easily create a Node.js application, which is automatically configured with third-party services that allow it to be deployed. From this point, the user should be able to easily develop and monitor the health of their live application.

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.

Related Work

There are a number of tools providing similar DevOps solutions or application creation for their users. We summarize the notable examples here. Spring Boot is a good example of project creation that does configuration work for the user. IBM Cloud and Google Cloud offer premium general solutions to DevOps and project monitoring. Our application is able meet a demand none of these other applications would be able to by focusing on simplicity for our users. Rather than supporting a dozen languages, a hundred third-party services, and a multitude of settings and customizations, we support a single language and a small set of third-party services in order to give our users a non-overwhelming, understandable tool to use.

Spring Boot

Spring Boot is an opinionated approach to the creating Spring Based Applications. The Spring Framework is a Java framework which makes it very easy to make web based applications. Spring Boot makes it easy to create applications in the Spring Framework by including third-party libraries and doing most of the configuration work. Spring Boot has the ability to make standalone applications, which already include a server embedded. It automatically does configuration and pom file generation, meaning it easy to get up-and-running quickly. Finally, it provides health checks and metrics for the application.

Through the addition of some of the open source libraries such as Zuul, Hystrix, RabbitMQ, and Eureka, Spring Boot has become the most common choice for creating a microservice architecture. Spring Boot does many other things such as providing ways to register applications with Facebook and Twitter and providing a uniform way to access data in Neo4J, MySQL, and MongoDB. Finally, there are easy ways to automate the configuration of Docker to make the Spring Boot application into a docker container.

This is only scratching the service of what this framework can do. We want to do some very similar things with hammer-io. However, our product will live in a completely different domain. Instead of Java, we will be targeting node.js developers. Below is a screenshot of the Spring Initializr, a handy means of generating a Spring Boot project on the web. We hope to provide a similar means of project generation for our users.

IBM Bluemix (now IBM Cloud)

IBM Bluemix is a cloud platform as a service (PaaS). It supports integrated DevOps with the ability to build, run, deploy and manage applications in a number of programming languages. Started in 2014, Bluemix offers the same sort of DevOps platform that we are aiming to create, with several notable exceptions. The first major difference between our product and Bluemix will be our focus on streamlined development of microservices in Node.js. While Bluemix supports the Node.js language, it doesn’t emphasize microservice architectures, nor does it bootstrap projects for you or enable the entire DevOps pipeline in a single command.

A second notable exception is that most of Bluemix’s functionality is locked behind a paywall. Our products, on the other hand, will be open-source and freely available for use.

Google Cloud Platform

Google's suite of cloud computing services includes application hosting, database hosting and analytics, along with a variety of other products related to cloud computing. Relevant to our product is number of supported DevOps options. Among these DevOps options is the Microservices Architecture on Google App Engine. This includes support for microservices for Python, Java, PHP, and Go applications. This most closely matches this project's long term goal of providing microservice support for JavaScript applications.

This platform provides an extensive dashboard for monitoring and managing deployed applications. For example, recent API requests, resources used, and documentation. An element prominently displayed on Google's monitoring dashboard, which will not be appearing on our page, is a billing component. Google's pricing model is pay as you go, with a modestly generous free trial first year.

Overall, Google provides an extensive platform for hosting and developing applications for their cloud. The user experience is generally overwhelming for an independent project or student; the target audience is late stage startups and businesses who are looking for a premium catch-all solution.

Deliverables

This project delivers the following products:

1. **Automated DevOps Process:** A tool to generate a NodeJS application, ready for our users to begin development and deployment of the application
2. **Deployment Monitoring:** An application to monitor the health and status of the deployed NodeJS applications

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 third-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.

This DevOps process should have all of the following major capabilities:

- 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

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 logs/reports, memory usage, CPU usage, application load, and uptime of each application.

Furthermore, this can be a place for users to manage their projects. This includes capabilities to create and manage their team, view current tasks related to its development, and view histories of its build and deployment through integrations with the third-party tools their application is actively utilizing.

Design

Requirements

Functional Requirements

- Automated Devops process for NodeJS applications
    - A command line tool to achieve the following:
        - Setup a new project (generates default configuration files) based on the services that a user wants to include. Supported options include:
            - Continuous Integration tool for build pipeline
            - Containerization tool
            - Deployment service
            - Source Control user wants to use for development
            - Web framework for the application
            - Testing framework
            - Object Relational Mapping framework
        - Download the default files for a new project based on the services that a user wants to include

    - Automate delivery of code to a continuous integration environment
      - Build the application
      - Containerize the application based on tooling options
      - Run test suites
      - Report status and results
      - Deploy the application to the selected cloud hosting provider
    - Ability to configure various services with a configuration file
  - A web app that has the same features as the command line tool
- Framework to develop microservice applications
  - Provides a command line interface to do the following:
    - Generate NodeJS templates for new microservices
    - Integrate new microservices into an existing microservice architecture
    - Configure the microservices (host, port, etc)
    - Ability to have features like API gateway to different microservices and load balancing out of the box
- Monitoring Platform
  - A web app that has the following features:
    - Authentication system to authenticate users
    - All the functionality of the devops cli tool - ability to generate project files, build the project, and deploy it based on the options selected by the user
    - View reports, results, and statistics about the DevOps process
    - Manage the build and deployment of services
      - View release history
      - Rollback to previous versions if needed
      - Deploy new versions
    - Ability to invite users to a project
    - Ability to connect account to third party services
    - Manage user account information
    - View statistics, reports, and analytics about the development process
      - Issue completion
      - Development time per issue
      - Code Coverage
      - Build information
    - View statistics, reports, and analytics about deployed microservices
      - Uptime statistics
      - CPU usage
      - Memory usage
      - Time for each request to complete
      - Endpoints accessed
    - Allow reports and logs to be downloaded

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) and Windows based systems
  - The system will support Node.Js version 8.x
- Reliability
  - The deployed web applications will run 24 hours a day, 7 days a week
  - Uptime for deployed web applications shall be >99%.
- Security
  - The system will not store any plain-text passwords in configuration files or elsewhere
  - All passwords will be encrypted
  - User information obtained from third-party services will be stored securely with the user's permission

Implementation Details

Technical Stack

Node.js

Node.js is JavaScript runtime which allows developers to write JavaScript outside of the web browser. It is built on top of the Chrome V8 JavaScript Engine; the same engine which runs the Chrome Browser. It is available on every major operating system, including Linux, macOS, and Windows. Node.js is an event-driven framework, which allows for asynchronous I/O. Node also includes the Node Package Manager (npm). npm is rich ecosystem which allows users to easily manage and include dependencies for their project. The npm ecosystem contains almost 700,000 packages which can be included in any project.

For more information on Node.js and the Node Package Manager, visit the following webpages:
https://nodejs.org/en/ https://www.npmjs.com/

Express.js

Express is a framework for developing web applications using Node.js. We used Express to define our API endpoints. Express works on defining a router to handle incoming HTTP requests. Each request has a corresponding controller where incoming requests can be parsed and processed. Express also makes use of a middleware pattern. Which allows a developer to essentially create a chain of actions that need to take place before the request can be handled by the controller. We used middlewares to write code to validate incoming HTTP requests to ensure that they were properly formed, meaning that no values were missing in the JSON and each value was of the right type. We also wrote middleware that checked authentication tokens. This allows us to write authentication code once and plug it in to any route that needed to be authenticated. Express can be installed via the Node Package Manager.

For more information on Express.js, visit their webpage https://expressjs.com/.

Sequelize

Sequelize is an Object Relational Mapping tool. It allows developers to simply define their database schema. For each table in the database, the developer creates an object that defines the value and the types for each column in the table. The developer can also reference other database tables and Sequelize will automatically create a mapping between the two. It will automatically create methods to allow the user to query the relationships. Sequelize generates the database based on these object definitions. Sequelize also eliminates the boilerplate code needed to query a database by automatically generating simple methods such as findById() and findAll(). Sequelize also has a consistent and simple way to write more complicated queries. Sequelize also provides ways to update the database schema safely, without needing to write database update scripts, making it easy to manage development and production databases. Sequelize can be installed via the Node Package Manager.

For more information on Sequelize, visit their webpage: http://docs.sequelizejs.com/.

MySQL and SQLite

MySQL is a database management system. It allows for persistence data store and access. We use MysQL to store almost all of our information, including user information, project information, and tool information.

SQLite is very similar to MySQL. However, we use SQLite as an in-memory database, meaning that when the application shuts down, all data is lost. We use SQLite for testing. We can then test our services which access data without mocking the data from the database. Using Sequelize makes switching between MySQL and SQLite very easy.

For more information on MySQL and SQLite, visit the following:

https://www.mysql.com/ https://www.sqlite.org/index.html

Firebase

Firebase is a cloud based NoSQL database storage. It works very similar to MongoDB. Firebase makes it really easy to stream data between two different services since it has a very robust SDK to access data. Our connection data write to Firebase happens in Koma. Koma collects the data from the user's application by using Skadi and then is written to Firebase in realtime. Yggdrasil then reads the data from Firebase in realtime and is then used to create the graphs, charts, and widgets on the Monitoring Page.

Fore more information on Firebase, visit https://firebase.google.com/.

Docker

Docker makes it easy to bundle and deploy applications into a consistent environment. Docker essentially creates a mini-environment for the application to live in. Docker allows the user to add whatever dependencies they would like. Everytime a developer builds the application, a new environment is created for the application to live inside. Docker only starts off with a small amount of dependencies and allows the user to add in whatever dependency they like. Docker allows developers to deploy their applications to any server and gives them confidence that their application will run since it is not using the environment of the server, but is using the environment of the container. This means that the environment of the application is consistent across development, staging, and production servers. We used Docker to package Koma, Yggdrasil, and Endor. This allows us to easily deploy our applications to any server that we needed to, without changing any configurations.

For more information about Docker, visit their webpage: https://www.docker.com/.

React

React is a javascript library for building user interfaces. It allows developers to build reusable UI components that can be used across the application with ease. The components are built using javascript and JSX. JSX adds XML like syntax to javascript. It looks very similar to HTML tags. Developers can write "view layer" of the component in JSX. Components also have the ability to receive data and maintain an internal state. The view can be defined based on the data the component receives or the internal state of the component. React efficiently takes care of re-rendering when the data changes. The entire UI in Yggdrasil is built using React. We also make use of a component library called material-ui that has a set of prebuilt react components styled based on Material Design guidelines. Material Design is a set of principles for designing user interfaces by Google.

Since being released by Facebook as an open source project in March 2013, React has seen an astronomical rise in its adoption across the industry. It has a vibrant community that is actively supporting and working to improve the library and the tooling around React. For more information, visit the webpage: https://reactjs.org/

Redux

Redux is an open source library that is used for managing the state of javascript applications. It makes state management simple and can be used with frameworks like React and Angular. It is inspired by the Flux architecture by Facebook for managing state in frontend react applications and by Elm, a functional programming language. It relies heavily on functional programming paradigms. Redux maintains the state of the entire application in a tree of plain objects and arrays. The amount of data that goes in there is determined by the developers. It also provides functions to access and update the state. The state cannot be accessed or updated in any other way. By maintaining a “single source of truth” and restricting the ways in which the data can be accessed and updated, redux provides many powerful techniques: keeping track of all the updates, ability to “time travel” through the history of updates and revert to a previous state if needed and view the state and UI at each point in time, allows easy multiple react components to easily access the data without having complex nested component hierarchies. React components can subscribe to data in certain parts of the state and if any other component makes updates that part of the state, the subscribing component will get the updated data and react can rerender accordingly. This allows developers to build scalable frontend applications without having to deal with the additional complexities of state management. We make use of Redux in Yggdrasil to manage the entire state of the frontend.

For more information on redux, visit their webpage: https://redux.js.org

Babel

Babel is a javascript compiler that is used to transpile next generation javascript code into browser compatible javascript code. Latest versions of Javascript has many functionalities that allows developers to write code that is less prone to bugs and is a lot more clean and concise. Unfortunately, many browsers don’t yet support all of it. Also, users on older versions of browsers will not be able to run web apps written in next generation Javascript. Babel allows developers to write next generation Javascript by transpiling the written code to older syntax of Javascript to make it compatible with most browsers for frontend apps and older versions of NodeJS for the backend apps. In addition, Babel can also transpile typed Javascript written using libraries like Flow and Typescript. We make use of Babel to transpile all our Javascript code written in ES6/7 across all the services (Endor, Yggdrasil, Koma, Skadi) with features like async await to browser/Node compatible code.

For more information about Babel, check out their website: https://babeljs.io

Design Overview

Our project is comprised of 5 different systems to help orchestrate the process of setting up an automated DevOps pipeline and a tool to monitor the user's applications. These applications are connected through the exposed APIs of each system and can communicate over HTTP calls.

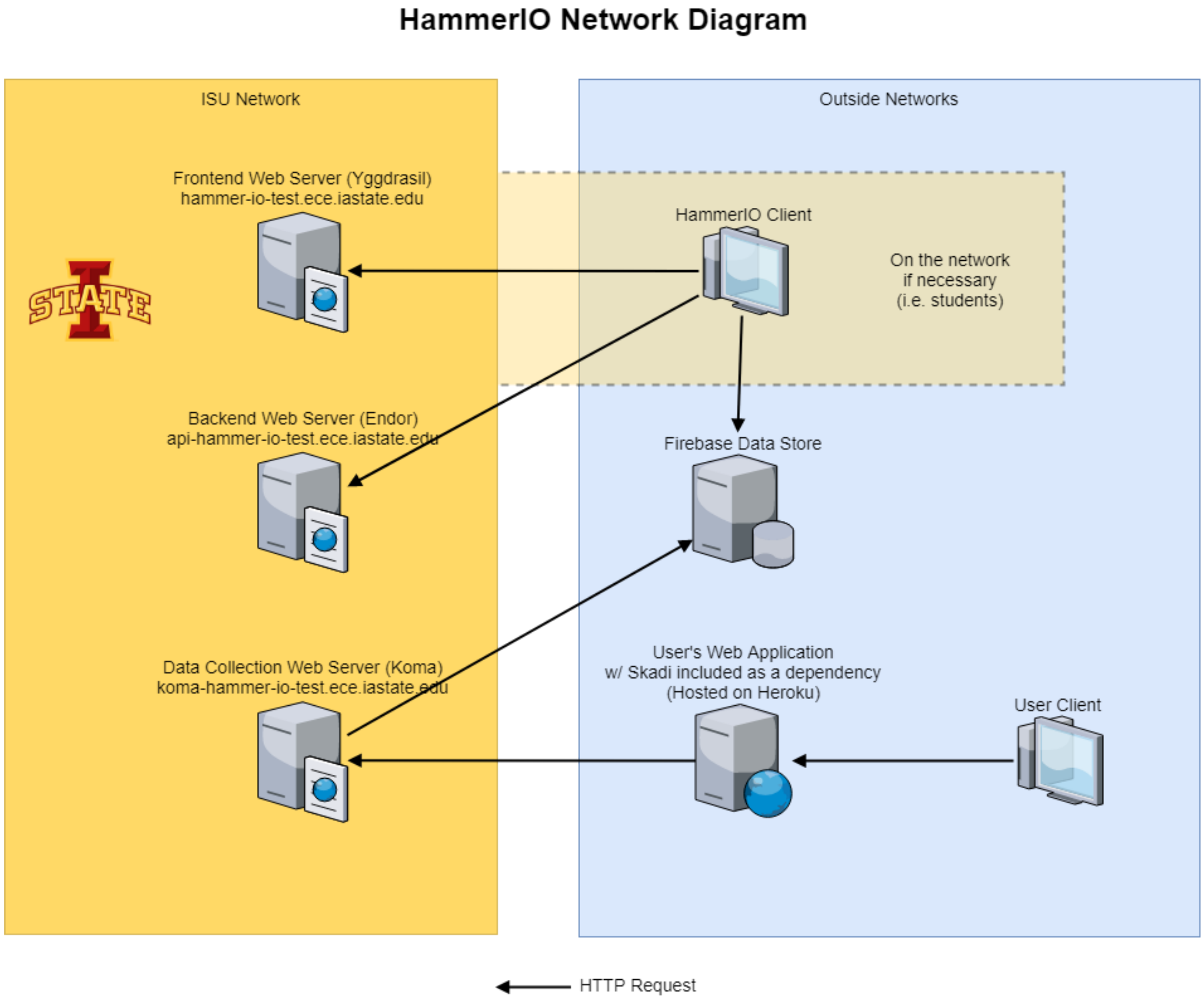
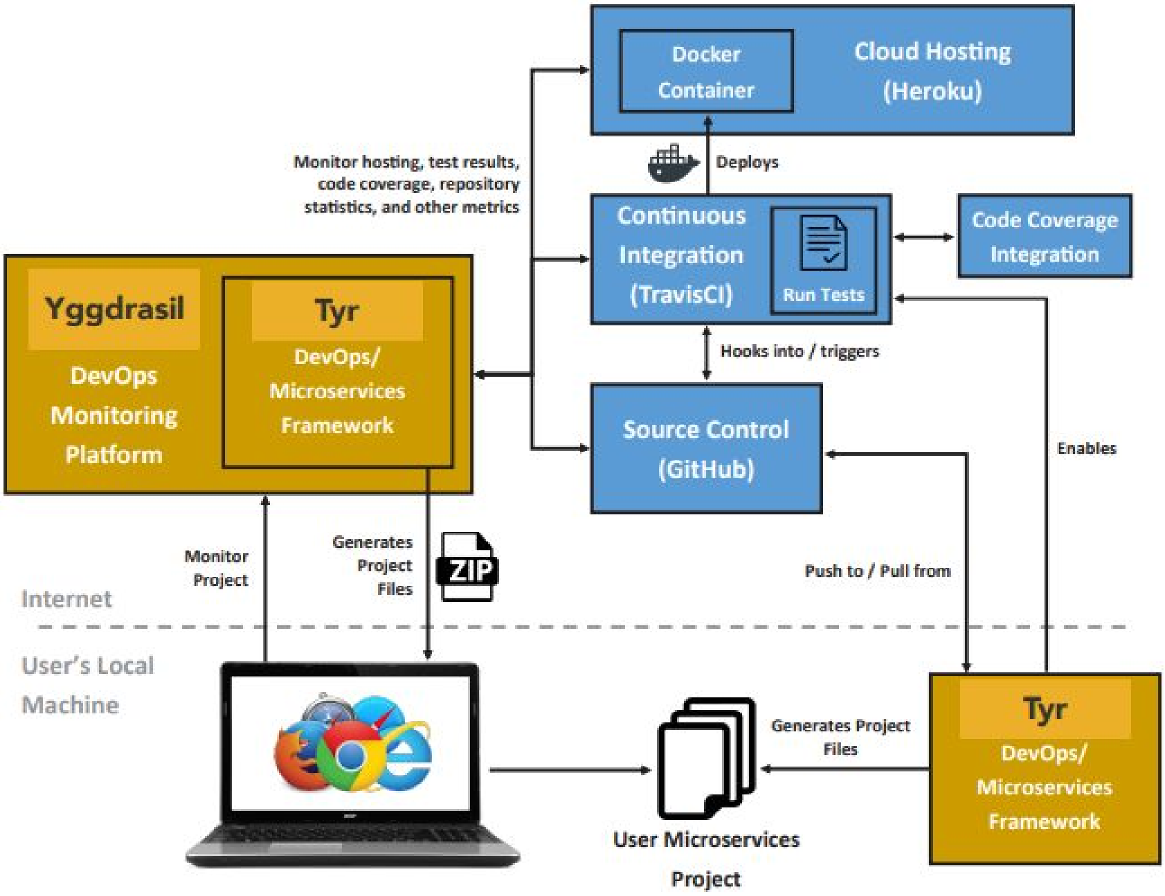

HammerIO Network Diagram
ISU Network
Outside Networks
Frontend Web Server (Yggdrasil)
hammer-io-test.ece.iastate.edu
HammerIO Client
On the network
if necessary
(i.e. students)
Backend Web Server (Endor)
api-hammer-io-test.ece.iastate.edu
Firebase Data Store
Data Collection Web Server (Koma)
koma-hammer-io-test.ece.iastate.edu
User's Web Application
w/ Skadi included as a dependency
(Hosted on Heroku)
User Client
HTTP Request

Figure 1: System Network Diagram

Figure 2 shows a component block diagram giving the general idea of how the different components connect and communicate with each other.



Docker Container
Cloud Hosting (Heroku)
Deploys
Monitor hosting, test results, code coverage, repository statistics, and other metrics
Continuous Integration (TravisCI)
Run Tests
Code Coverage Integration
Yggdrasil
DevOps Monitoring Platform
Tyr
DevOps/ Microservices Framework
Hooks into / triggers
Source Control (GitHub)
Enables
Monitor Project
Generates Project Files
ZIP
Push to / Pull from
Internet
User's Local Machine
Generates Project Files
Tyr
DevOps/ Microservices Framework
User Microservices Project

Figure 2: Component Block Diagram

In the following sections are more detailed design features and functionality for each of the systems.

Yggdrasil

Yggdrasil (Yggdrasil is an immense mythical tree that connects the nine worlds in Norse cosmology) is the frontend application. It addition to being the microservice monitoring platform, it also has all the functionalities of Tyr - the command line tool for project creation. Yggdrasil primarily interfaces with Endor, the backend, with some interaction with Github and Heroku for connecting third party user accounts to our system. Endor maintains all the data about users, projects, etc and is the REST API for Yggdrasil.

Yggdrasil is built using React and Redux. React is used for the user interface, while Redux is used to manage the state of the frontend application. We use Babel to transpile all the next generation javascript code that we write to browser compatible code. The entire application is containerized using Docker and deployed to Iowa State University's servers.

Routing

Yggdrasil is a single page application. This means that we only serve the html file once when the user make an initial request. After that all the routing is handled on the frontend. We use a library called react-router to handle the routing. For each route we specify the respective react component to render. This component can have multiple components within itself. The routes on the frontend are listed below.

/home

This is the page where users first land when they log in. If it’s a new user, the user will see an onboarding carousel that describes the features of the application and a prompt to create a new project. If the user already has projects, then they will see the projects they own or projects in which they are contributors.

/tyr

This page provides details about the command line tool Tyr. It gets all the information from the README.md in the Tyr code repository on Github. If the information is updated in the README, it will be reflected here as well.

/settings

In this page the user can update their account information, manage invites to projects, and connect their account to third-party services in order to be able to use them in their projects. For connecting to third-party accounts, the user is redirected to the respective OAuth2 login pages of those services. This ensures that we do not have access to the login credentials of the user from those third-party apps at any time.

/projects/new

This is the page where users can create new projects, similar to the Tyr. Users can select the services they want to use in their application. Certain services requires connecting their account to those third-party services. This is done in the settings page. If the account is not connected, then those options are disabled. On successful creation, the user can download the project files to their computer and is redirected to the project monitoring page.

/projects/{project_id}

This is the page where users can view information about the specific project (determined by the project id in the URL), invite and remove members from the project if the user has the required privileges, monitor the statistics of the deployed application, and obtain their Koma API key to use for the project monitoring functionality.

Redux usage

We make use of redux to manage the entire state of the frontend app. Redux has two important concepts - actions and reducers. Actions are plain objects that contain a field called type which is a string that defines the type of action and a payload which is the new data that is needed to update the state of the redux store. All the changes to the state happens only using actions. The reducers intercept these actions and based on the type of action updates the redux store as needed. The redux store maintains an almost identical copy of all the data on the backend. Each time we make a GET request, data is added to the redux store. Similarly, the redux store is updated based on the type of the request (POST, DELETE, PUT, PATCH). If multiple react components are listening to the same part of the state from the redux store, and if one of them changes the data, the others will be notified of the change and the components will rerender with the new data. This model scales really well in much more complex web apps with hundreds or thousands of components and many things happening simultaneously on the page. Since all the backend data that is fetched is stored in the redux store along with the state of some complex UI interactions, if a page fetched some data from the backend and stored it in the redux store, the data is available for the next page that requires it without having to make a request to the backend. In this manner we avoid the network overhead of making multiple requests for the same data. In addition, we use a library called redux-orm. This library allows us to normalize and maintain all the data in the store referenced by ids. This makes lookups much faster, and is similar to a hashtable.

Major Features

Project Generation

Similar to Tyr, Yggdrasil supports the ability to create new projects. The app walks the user through the list of options and the user can pick whichever ones they want. Options like TravisCI and Heroku require the user to connect their accounts to those services. Once the user has selected all the options, the frontend will send all the information to Endor, and Endor will create the project in the Database, generate the files using Tyr by passing the configurations to it, and will then serve the files to the frontend. The user can then download all the files from the browser. If any third-party service was picked, then Tyr will also take care of setting up the project on that service. Tyr makes use of the third-party API's to accomplish it, which is accessed using API tokens. Each service has its own token. These tokens get stored in Endor when the user connects their account to third-party services from the frontend.

Project Monitoring/Management

The ability to monitor and manage projects is unique to the web application. If the user selected a source control tool, then the project page will display the newest issues. If a continuous integration tool was selected, then the web app display the build status of the last build. If the user chose a cloud deployment option like Heroku, then the app would be deployed to the cloud and the frontend will display statistics about the deployed app. The main statistics displayed are uptime, memory usage, http endpoints of the app that were accessed, and time for requests to complete. This data is fetched from firebase in real time. The deployed app is sending this information from the app using Skadi and Koma which will be discussed in a later section. The frontend maintains a socket connection with firebase and listens for streams of data. As new data comes in, the UI is renders it accordingly.

Project users can also add new members as owners or contributors to a project. They simply need to know the username of the user they want to add. In addition, owners can remove users from the project. Yggdrasil makes use of the Endor API for all these changes. An owner of a project can also view the API key for Koma that has to be placed in the generated project to allow it to send monitoring specific data to Koma.

Account Management

The user can also manage their Hammer account from the web app. They can update their information, all of which is stored in Endor. They also have the ability to manage their invites to projects and accept or decline invitations to join projects.

In order for the user to connect with third-party services like TravisCI and Heroku in their project, they must first connect their accounts with those third-party services. This is done on the user settings page. To connect to any one of these third-party applications, the user is redirected to a permissions page on the third-party app, in which they agree to share certain information with our application. They must log into the third-party website to verify their identity. Our system does not have access to the third-party credentials. Behind the scenes, this process gives our application a key which is exchanged for a special API access token linked to this user, which the frontend sends to Endor for storage. This API token allows our application to automatically perform actions in the third-party application on the user’s behalf; for example, getting the latest build histories.

Endor

Endor is the backend microservice. It unifies a lot of the microservices we have. It is the backend for Yggdrasil, so projects are created through Endor, which stores project information along with which third-party services they use to later retrieve more information about the project. To generate projects, Endor uses Tyr as a library. Endor also communicates with Koma about the creation of a new project.

For Endor, we used layers to separate functionality, and inside those layers, we further divided up the functionality into components. This made it easier to reason about the project and locate where certain functionality occurs. The layers follow the MVC pattern. The first one, the routes layer, is the view since it is the point of entry into the application. The controller is the controller layer as it controls the data flow between the services and the routes and controls what errors are returned to the user. The model layer is the services layer. These manage the data access from the database.

The components are split based on what parts of the application they affect. Some of them, like HerokuAuth, GitHubAuth, and TravisAuth all make requests to third-party services to retrieve OAuth2 tokens are store them. Auth is another different component because it stores the tokens for a user and exchanges tokens for a username/password. The rest of the components are responsible for a table in our database.

We chose to use a MySQL database to store our data. We could have used a MongoDB which would have paired nicely with Node.js, but the team was already familiar with SQL, and sequelize, the Object Relational Mapper we used, made using MySQL a breeze. Plus, our data was very relational, which is one area where MongoDB does not excel.

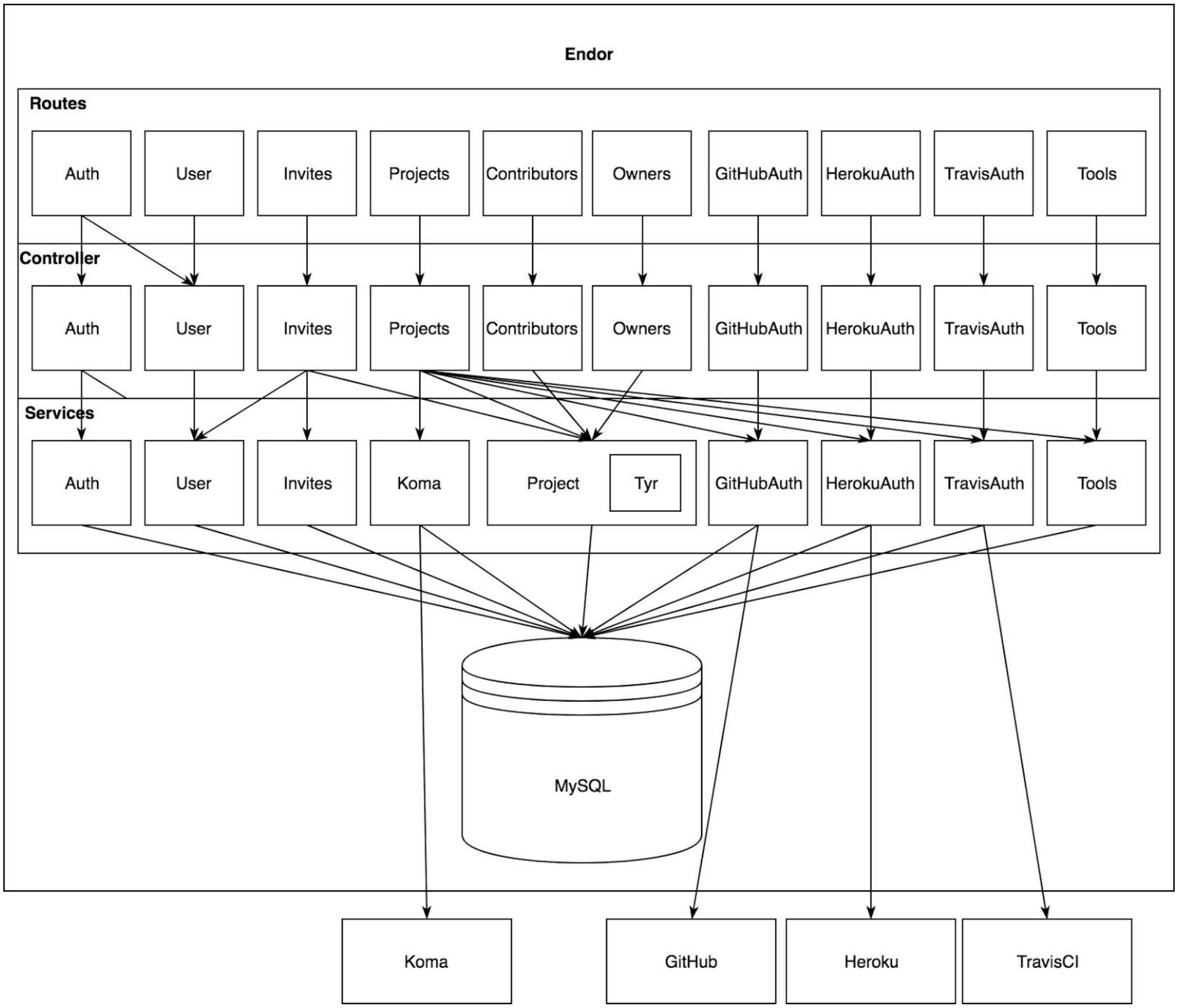


Figure 3: Endor Component Diagram

Routes

The API endpoints are segmented based on their functionality. Almost all of them, except when registering a user, require the user making requests to be authenticated with either a token or a username/password. If they are related to updating or deleting a user, they must have access (for example, they have to be the user) in order to perform the action. If they are trying to update the metadata on a project, they must be an owner on the project. Requests are also validated here to ensure the data is correct and the request is formatted in a way the controllers will be able to process.

To learn more about the routes visit: http://api-hammer-io-test.ece.iastate.edu/

Controllers

The controllers process requests and organize data returned from the services to be sent back to the user. Most of the error handling happens here since if an error is thrown, nothing else should happen and an error, not necessarily the same error that occurred, should be reported to the frontend.

Services

The services in Endor control database access. They perform CRUD operations on the database, and validate the data before it enters the database. If an item should be encrypted or hashed, it would occur in the services right before it enters the database. Some of the services also make calls to third-party services to get more information about a certain project, or to create an OAuth 2 token for a user.

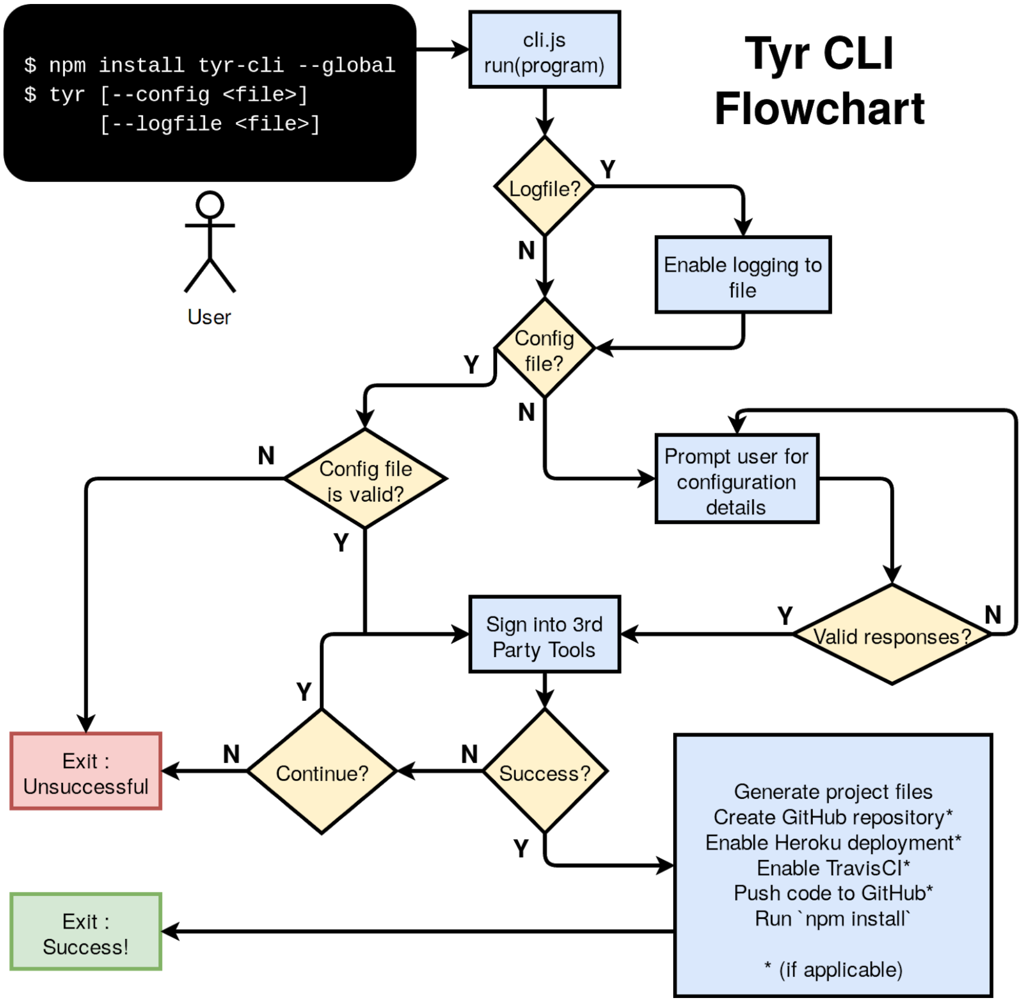
Tyr

Tyr is a tool to generate a project, complete with a deployment pipeline and data monitoring. It can be used through its CLI and also has functions available for other applications, like Endor, to use. Tyr supports various third-party tools. Currently, it supports:

- Source Control
- Continuous Integration
- Containerization
- Deployment
- Web Framework
- Testing
- Object Relational Mapper
- Config Storage (this is default to Node-config)

We designed Tyr to allow it to be used first as a CLI, then later redesigned parts to make it usable as a library as well. Although we only have 7 different categories for add-ons to a project, we plan to allow for more, and designed the code in a component-style to allow new types of tools and implementations of tools to be added quickly. To add a new component, all that needs to be added, is the component itself, the command line interface view, and a mapping from the component's name to the component's functions in TyrLib. Then with the addition of tests and updating the validation of a .tyrfile, the new component will be ready to ship.

The command line for Tyr is easy to use. It needs to be downloaded through NPM and it will follow the flowchart below. It starts by determining if a file was passed in or not, if it was, and the file is valid, it begins generating the project. Otherwise, it will prompt the user for the information to fill out the config file. This is information like the project name, the version, the license, the authors, which tools they want to use for each category, etc. Then it begins to generate the project. This usually takes 2-3 minutes as Tyr waits for TravisCI to be linked to the project repository that was created on GitHub. Otherwise, the project would not deploy straight through to Heroku the first time it was deployed.



```
$ npm install tyr-cli --global
$ tyr [--config <file>]
      [--logfile <file>]
```

Figure 4: Tyr CLI Flowchart

Tyr can also be passed a .tyrfile to create a project. The .tyrfile contains the project information like name, version, authors, etc., and which tool the project uses for each of the seven categories. We did this to make it easy on developers to create a project similar to one they've created before. We also designed this .tyrfile to make updates to the project's configurations later in the lifecycle of the project, and we needed a way to keep track of what configurations the project was currently using. The .tyrfile solved both issues at once.

In the future, Tyr should be able to support a variety of types of applications, and a variety of options for each type of tool. However, all of these tools will interact differently with each other so we tried to design it so each tool is as independent as possible. For example, Docker can be deployed anywhere, while TravisCI only works with GitHub. Docker's deployment then cannot be tied just to Heroku, and if we were to add an if statement on what to do if heroku is chosen, we would have to add another if statement for each other deployment platform we add to the tool. This would be inconvenient and ugly. We try to make it so that if there would be an if statement, we abstract it to iterate over the tools chosen by the user to use. Then we plug each tool name into a map that has the name of the tool tied to a function that performs the necessary behavior for that tool. This makes it easy to add additional tool support as described earlier.

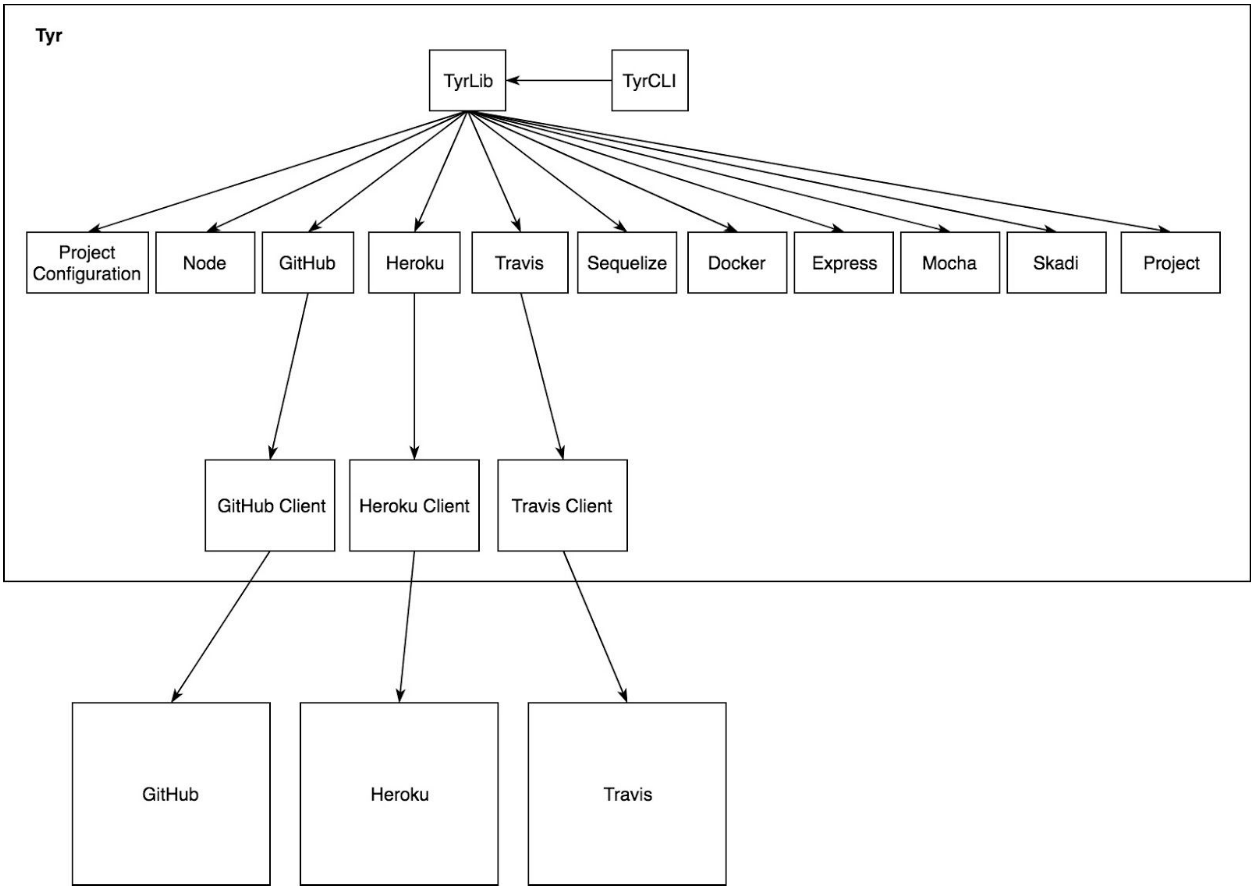


Figure 5: TyrCLI Block Diagram

Another example where we planned for independence is when creating the package.json. We don't know what dependencies to add to the package.json without looking through each of their tools. To solve this, tyr first creates a bare bones package.json, and each tool service adds its necessary dependencies to the package.json.

TyrCLI

TyrCLI is the cli entry way for the application, it contains the views, validation and logic behind the CLI. Then it passes the information to TyrLib.

TyrLib

This is another potential entry way into the application, it has several exported functions available to other applications. It orchestrates all the different services that are being used in the application.

Services

Most of the services contain functionality to generate static files for the tools and potentially call other clients to make requests to third-party applications.

Clients

These clients handle the communication between Tyr and third parties like GitHub, Travis, and Heroku. They make the requests and do a little error handling, but for the most part hand their errors back to the service that called them to see if the error can be fixed.

Koma

Koma is a standalone Node.js server used to aggregate application monitoring data. It was designed using the Express.js web application framework. The API allows a client to send information about application heartbeats, operating system data, and http request/response data to the server. This data is then aggregated and stored in a Firebase Realtime Database. Authentication data, such as project ids and API keys, are encrypted and stored in a MySQL database.

Koma's main goal is to collect data sent by a user's application (via Skadi) and store it for use in monitoring in the main Hammer-IO web application Yggdrasil. Any projects created via Yggdrasil have Skadi automatically incorporated into them. Skadi sends runtime data about the project to Koma. Skadi is currently the only client for Koma. However, in theory, Koma could be extended to any number of clients. The component diagram for Koma is shown in Figure 6.

API Endpoints

Koma is comprised of three API endpoints for posting data and one for authenticating new projects[1]. The data endpoints handle POST requests for heartbeats, HTTP data, and operating system data. All endpoints require an API key, which is given to the project when it is created on Yggdrasil.

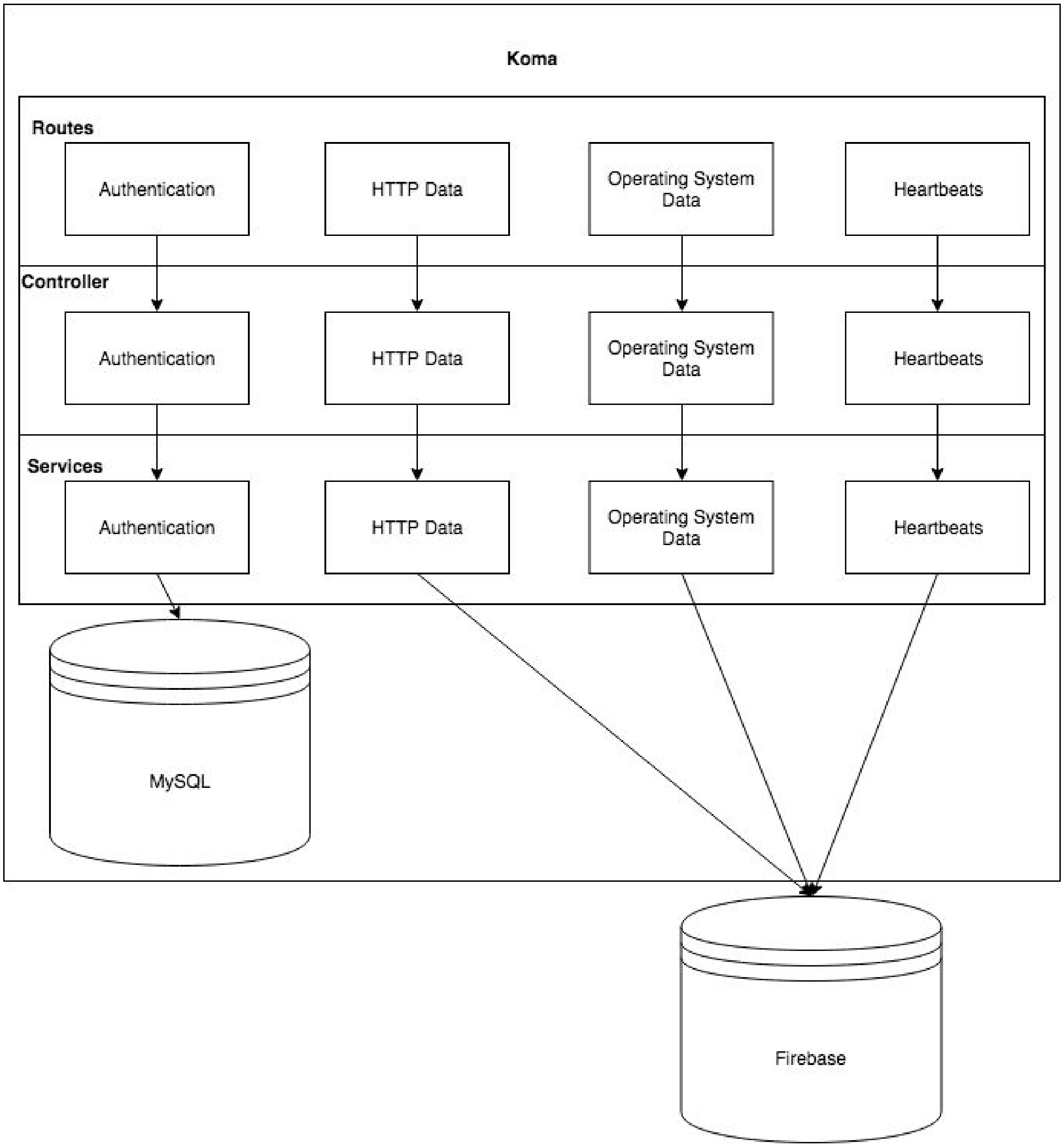


Figure 6: Koma Component Diagram

[1] *Project* here specifically refers to a user application created through Yggdrasil.

POST /heartbeats

The heartbeats endpoint accepts POST requests sent at periodic intervals. Upon receiving a heartbeat, Koma calculates the current timestamp and posts this timestamp to Firebase. Heartbeats provide evidence that an application is up and running. They are used by Yggdrasil to calculate the health of the application—that is, how much time the application has been up or down.

POST /httpdata

The HTTP data endpoint allows projects to send information about the HTTP requests and responses that are coming through their application. This endpoint requires the project to send response time information, size of the data, HTTP status of the response, and more. The data is then sent to Firebase for storage and consumption by Yggdrasil.

POST /osdata

The OS data endpoint allows projects to send information about the operating system that they are running on. Memory Used, Free Memory, and Percentage of Memory Used are sent to Firebase for storage and consumption by Yggdrasil.

Authentication with Endor

Since this service is separate from Endor, we needed a way to authenticate Koma project creation and authorize access to Koma data without duplicating the existing mechanisms in Endor. We accomplished this by creating a handshake authentication mechanism. Endor sends the id of the project to be created on Koma and a secret key. Koma then looks at the secret key to determine if it came from Endor. If the authentication was successful, then Koma creates a new API key for the project, stores it in its database, and sends the key back to be stored with the project information in Endor. The API key is accessible to the project owner(s) on the Project Settings page in Yggdrasil. The key is then used to configure the user’s application to authorize that application to post data to Koma.

Operating Environment/Deployment

Koma is a standalone web server, which is deployed to an Iowa State Ubuntu Virtual Machine. The application runs within a Docker container. Figure 7 shows a diagram of the deployment environment for Koma.

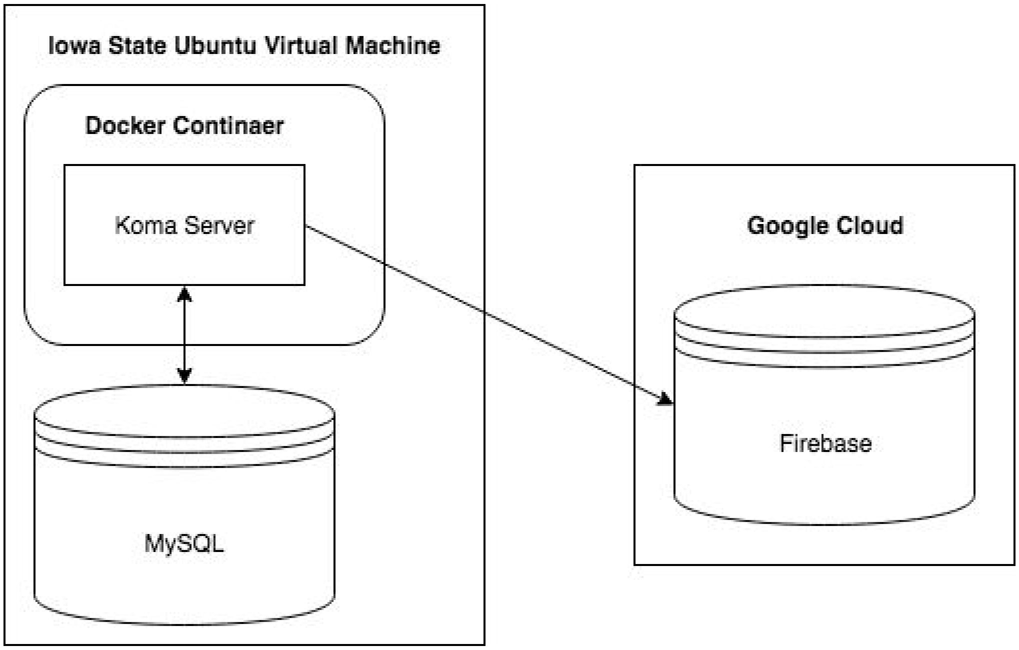


Figure 7: Koma Deployment Diagram

Design Considerations

One of the major design decisions that we needed consider for this component was scalability. In addition to HTTP and OS data being sent frequently, monitored projects can send heartbeats as often as they want. The polling interval is not limited. The service has the potential to handle a *lot* of data. Koma could have been easily put inside of Endor; however, if we did that, we may have bogged down Endor as usage increased. This would have made all of the other functionality of they system slower. By separating Koma from Endor, we were able to keep both Koma and Endor more performant. Also, by keeping them separate, we are able to scale them independent of each other. When Koma needs more memory or more servers, we only have to adjust Koma without impacting Endor. We also needed to ensure that both Endor and Koma were as reliable as possible. By decoupling the servers, when Koma or Endor goes down it does not impact the other server. This means that if Koma went down, users would still be able to access the system (it just wouldn't be collecting new data).

Another important design consideration that we made was choosing to use Firebase Realtime Database to store monitoring data. We needed to be able to display data from the user's project in real time for application monitoring services. We could have created something with websockets to handle this; however, it would have created a lot of overhead and load on the server and drastically slowed down feature development. By using Firebase, we are able to rely on Google's infrastructure to store the data reliably and securely while keeping it scalable.

Skadi

Skadi is a Node.js module that streams data about a project to Koma to be aggregated for monitoring purposes. Skadi can be installed via npm, however it is automatically installed in any project that is created via Yggdrasil. It lives as a dependency in the user's project. Heartbeat and Operating System information is sent at periodic intervals, while HTTP request and response data is captured via Express Middlewares.

Data Collection

Skadi's main purpose is to facilitate data collection. This allows us to monitor the status of the user's project in real time.

Heartbeats are an important part of the data collection process. Heartbeats are sent from Skadi to Koma. These Heartbeats provide a way to determine if the application is running or not. An application will not be able to send a heartbeat if it crashes, so "missed heartbeats" can be interpreted as a down or crashed service. Every *n* seconds, Skadi sends a heartbeat to Koma. The number of seconds is user-configurable via the .skadiconfig file.

OS data is also collected from Skadi. User's may want to know how much strain their application is putting on the system on which it is deployed. Every *n* seconds it transmits the following information:

- Free Memory
- Total Memory
- Percentage of Used Memory

This data is captured using the standard Node.js OS library.

HTTP data is also collected from Skadi. A user might want to know how many requests are being sent to their server or how many bad responses they are having. Skadi sends the following information to Koma for each HTTP request:

- Request URL
- Request Size
- HTTP Method (GET, POST, DELETE, etc.)
- Response Status
- Response Time

This data is collected for every request that is made to the user's server. This works by using Express Middlewares. Middlewares perform an action after a request is received and before a response is sent (hence, they sit in the "middle" of the request handling process). It captures the data needed from the HTTP request and response objects and then sends that information to Koma.

Design Considerations

When developing this library, there were a couple important issues that we needed to consider. We needed a way to collect the monitoring information with very little user impact. It shouldn't be something the user has to figure out and enable; it should be enabled and functional by default. To accomplish this, we integrate Skadi in the project creation functionality of Yggdrasil. This allows us to setup the user's data collection for them. We also wanted to make sure that this data collection library did not impact any business goals of the user's application, such as uptime or performance. To that end, the module is designed to make asynchronous calls with small payloads so the performance to the user's application is negligible. Additionally, we made sure that if Skadi fails for some reason, it will fail quietly and not crash the user's application.

Development Environment

IDEs and Editors

For IDEs and editors, we mostly used JetBrains projects. JetBrains provides excellent IDEs with strong support for JavaScript and Node.js application development. Team members either use IntelliJ or WebStorm, depending on their preference. Both tools give the same JavaScript support. They provide nice features like autocomplete, syntax highlighting, and syntax error reporting. Additionally, it gives helpful error reporting, like symantec reporting. For example, it can detect if a function does not exist on an object being used. IntelliJ and WebStorm also have an ESLint integration plugin for live feedback of linting errors. This makes it easy to fix linting errors before code is tested or committed. These features greatly speed up the development processes.

For database tooling, we use JetBrain's DBGrip and Microsoft's MySQL Workbench. Both of these tools allow developers to view and update data or schemas on the fly. In addition to query building, they also provide GUI's to directly edit the data in a table.

When a lightweight editor is needed, we use emacs, Atom, or vim (depending on user preference). Atom provides a more fluid interface, but emacs or vim are necessary for making script or configuration changes in the headless server environment.

Operating Systems

Our project has been developed on all three major operating systems: Windows, Linux, and MacOS. The cross-platform development is enabled in large part by the nature of the Node.js language and runtime environment. Running the applications in different operating systems also allows us to have greater confidence in the application's robustness across different user groups.

Development Process

Agile

We tried to follow the Agile best practices as closely as made sense for the team size and time frame. We did one week sprints. During each meeting with the client/adviser we gave a demo, had a retrospective, and then planned for the upcoming week. During this time, we also refined the backlog and started estimating how much time each issue would take. Issues were then assigned to a developer based on time estimation and the developer capacity. If a story changed or a critical bug surfaced during the sprint, we adapted to these incoming changes, quickly wrote issues for them, and then assigned them to a developer who was able to get them done. This process worked very well for us. It gave us weekly goals to accomplish, which helped keep the project moving forward. The process also provided quick and frequent feedback from the client during our weekly meetings.

Feature Branch → Pull Request → Code Review Workflow

We used a *Feature Branch → Pull Request → Code Review* workflow for development. This is one of the more popular workflows in Agile and works very well when there are clearly defined issues. For each issue being worked, the developer creates a feature branch off the current HEAD of the master branch. The developer then makes commits to the feature branch as they work on the issue. When the work for the issue is completed, the developer then opens a pull request and asks someone to review the proposed code changes. When this pull request is opened, a code coverage tool runs to verify that the commit diff has been covered by a test case. In addition, the test suite executes to ensure that no existing tests have been broken. Finally, a manual code review is required. During the manual code review process, the reviewer confirms that the desired functionality was implemented correctly and with good design. This rigorous process helped ensure we adhered to good design practices and kept the codebase in good condition as it continued to grow.

Code Coverage and Linting

We used code coverage tools and linting to help maintain the quality of the codebase. For Code Coverage we used a tool called Codecov, which hooks in with the GitHub repository and the continuous integration framework. Whenever a pull request is made, Codecov waits for the build to be finished. The build sends coverage information to codecov, which then makes a comment on the pull request in GitHub. We did not have a Code Coverage target goal, but we used code coverage to let us know where our coverage gaps were and to help “gamify” the sometimes tedious task of writing tests.

We also used a linting tool to help maintain a consistent styling across the different subsystems. We choose ESLint, a popular linting tool in the JavaScript ecosystem. Specifically, we used

Airbnb’s JavaScript style guide, which ESLint can automatically set up. Linting tools helped keep our codebases readable, maintainable, and organized. We had ESLint setup in our pre-commit hook, so code that did not adhere to this style guide couldn’t even be committed. Furthermore, the linting was integrated into the build process, so if somehow code that didn’t adhere to the style guide made it into the repository, the build would fail and prevent merging a pull request to master.

Continuous Integration

We used TravisCI for our continuous integration environment. TravisCI is an online tool that connects to a user’s GitHub repository. The developer selects the repositories they want to connect, and TravisCI will start watching those repositories. The developer then creates a *.travis.yml* file, which defines what should happen each time the repository is built. For example, you can tell it what tests to run, where to push code coverage results, and where to push the application after it is built and tested. The developer can also select what to do on build failure or on build success.

We used TravisCI to run tests, run linting, and send code coverage results to Codecov. On each commit, TravisCI does this. TravisCI has a GitHub plugin which shows the results of the build next to the commit; a red “X” for error and a green checkmark for success. TravisCI also does this for Pull Requests. It helps ensure that broken code is never committed to the master branch and that broken code is never deployed. In the future, we plan to have TravisCI build our Docker containers and push them to DockerHub, where we can set a hook to automatically deploy them. We currently have TravisCI working for continuous integration; however, in the future we would like to use it for continuous deployment as well.

Testing

Types of Testing

Throughout the development of this project, we performed various types of testing to ensure that the product met the functional and non-functional requirements.

- **Unit Testing**
  - Unit tests were written where feasible for all code that was committed into the repository.
  - The author of a new feature or bugfix was responsible for writing unit tests for his or her code, and the code reviewer held the author accountable.
  - Tyr, Endor, and Koma have unit tests for each component of the code
  - Yggdrasil is devoid of unit tests due to resource constraints. The limited time available coupled with the difficulty of testing UI code made unit testing infeasible.

- **Integration Testing**
    - Integration tests were written after multiple individual components were integrated to ensure they worked correctly together.
    - Developers decided amongst themselves on the set of integration tests each of them wanted to work on.
    - We also have integration tests to ensure our system works correctly with third-party services. The tests run very slowly because they must make HTTP requests across the internet, so they are not run with the regular test suite. Instead, they run periodically on TravisCI.
- **Manual Testing**
    - Manual testing was required to account for features hard to test with unit and integration tests. This includes:
        - UI features
        - Some integrations with third-party services
        - Ensuring our system works correctly when all our services are put together
        - Testing workflow of the app in an end-to-end manner
    - Whenever a new feature or bug fix is completed and a Pull Request is opened, reviewers pull down the changes and run everything locally to ensure everything works correctly before merging the new code into master. This manual testing process gave an acceptable level of confidence in the correctness of the implementation and in the minimization of bugs.
- **Acceptance Testing**
    - Manual testing by developers and weekly client demos were used for acceptance testing.
    - Continuous feedback received from developers during the development process and feedback from the client during weekly demos allowed us to reiterate multiple times in an agile manner and adhere to the evolving requirements for the product.

Testing Strategy

Testing begins locally on the developer's machine. At 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 tests to ensure that the functionality of other parts of the code base have not changed unexpectedly. If these tests pass (and if linting passes), the code is pushed to GitHub, where the tests run again to ensure that no changes were unexpectedly made while merging and to ensure that the code on GitHub is functional. This happens in a Linux environment using TravisCI. If the build fails on Travis, the code cannot be merged into the master branch on GitHub. Developers will be notified via email if the TravisCI build fails.

The code can only be merged once all the tests pass on TravisCI and after another developer has approved the pull request. This workflow ensures that no new bugs are introduced in other parts of the codebase that are unrelated to the new feature. Also, by having another developer pull down the new code and test it, we introduce an additional layer of systems and acceptance testing.

Once a feature has been approved and merged to master, the code can be deployed (for web apps) or published to npm (for npm packages). 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 new features and bug fixes. Otherwise, the developers will need to tweak or redesign and reimplement the current solution based on the client's feedback. Figure 8 illustrates this workflow. For more information about the build status of each service, please visit the following links:

Tyr	https://travis-ci.org/hammer-io/tyr
Endor	https://travis-ci.org/hammer-io/endor
Yggdrasil	https://travis-ci.org/hammer-io/yggdrasil
Koma	https://travis-ci.org/hammer-io/koma
Skadi	https://travis-ci.org/hammer-io/skadi

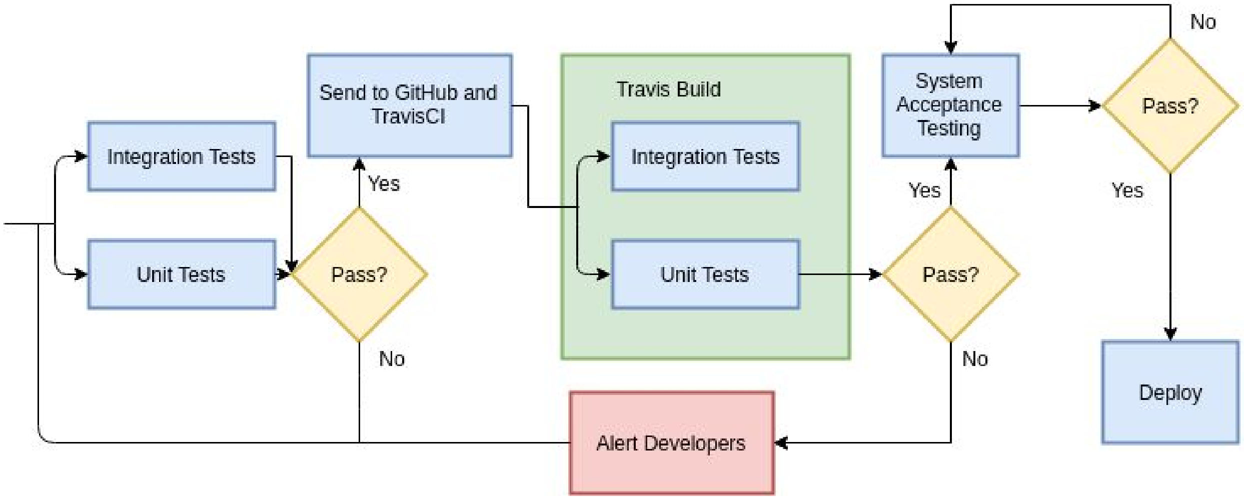


Figure 8: Testing Workflow

Testing Libraries

Mocha

Mocha is a javascript testing framework for Node.js. All our tests are written with Mocha. It allows us to organize the tests in tests suits based on each component being tested. It also has good test result reporting functionalities, which makes it easy to understand which tests and assertions failed.

Chai

Chai is an assertion library for Node.js that can be paired with many testing frameworks. It is able to perform more complex assertions on objects and arrays than Mocha. Chai has various tools to allow developers to write tests in the BDD or TDD style, depending on what type of testing style the developers prefer. It also has many additional plugins that allow for more powerful testing methods.

We use Chai in combination with Mocha to write integration tests. Chai has a plugin called chai-http which allows us to test the API endpoints. It can start the application, send requests to the endpoints, and assert against the returned responses. This enables us to write effective integration tests for the backend web server functionality.

Challenges

Novelty

Our product is not unique in its market, so various competitors will be challengers for our product. Some of these tools are professionally developed and maintained, so our product must be especially well-suited to a specific need that these other DevOps and automated development tools are not meeting. Our application rises to meet this challenge by limiting itself to support a single language, and only a small handful of third-party applications. The result is an overall application and experience that is incredibly easy to use and lends itself to be used in the classroom far more than premium alternatives which require significantly more overhead.

Third-Party Services

Our application relies heavily on third-party applications to perform many of the essential steps in the DevOps process. This reliance created a number of challenges for our group. In one such example, the TravisCI API is transitioning from V2 to V3. In the course of this switch, one function essential to our process is being deprecated and will no longer be supported. This means that today we are able to exchange a Github Token for a Travis CI Token, but this functionality may be removed tomorrow. We have prepared a work-around in which the user will be redirected and asked to copy and paste their key from a webpage during the Travis CI third-party integration step on the account settings tab. This specific challenge shows how using a large number of third-party APIs forces our application to be vigilant in both its testing and response time.

Security

Passwords

When a user creates an account, they create a username, which must only contain letters and underscores, and a password, which must be 8 characters long, and include one letter and one number. The username and password are sent to the backend, where the password is hashed and salted before being stored in the database. When a returning user signs in again, the username and password combination are sent to the backend. The password is then hashed and salted to be compared with the password stored in the database. The bcrypt library is what we use for encryption, since it is one of the most popular slow password storage algorithms in use today and because it is safer than trying to implement our own encryption.

Tokens

In addition to using a username/password for authentication, a user can also use a token. A token is created both when a user registers for and signs into the application. The session token is sent to the server on behalf of the user for every request when they are logged in, since all endpoints other than registration require an authenticated user. Upon logout, the token is destroyed to signal that the session has ended.

We also have a token that gets shared between Koma and Endor. When Endor creates a project, it makes a request to Koma to store the id of the project in Koma. Koma then verifies the token and generates an API key for the project. This token can also be viewed in Yggdrasil (for project configuration purposes), but it is only visible to project owners.

We also store OAuth2 tokens that are used to generate a project for a user in our database. We encrypt these tokens since we will need to use them again, but we don’t want them to be viewable. We use crypto, a Node.js encryption library, to encrypt the tokens for later use.

API Key

Skadi is placed in each project to monitor the status of the project. When the project is created, the Koma API key is generated for the project. The project can then use the API key to authenticate with Koma and send data to Koma to be stored in Firebase. Koma identifies the project and authorizes access via the API key.

Access Control

Each project can have owners and contributors. Owners can modify metadata about the project, such as who other owners and contributors are. The project appears on the dashboard of both contributors and owners, and both can see the data about the project. However, projects are not private, so they can be viewed by any user. Access Control is also implemented for users. Although users can view another user, only the authenticated user can update his or her information.

Conclusion

Lessons Learned

Through this senior design project, we learned a lot about working on a team and working extensively on a continuous project. This experience was valuable even in comparison to our internships because the project lasted longer than three months and a lot changed over the course of the project. Plus, we were involved in every aspect of the project, including design, testing, code reviews, and documentation. We gained a lot of experience in these areas and learned from the many mistakes we made.

At the start of the project, we made a lot of mistakes that became obvious with time. We started out with a vague idea of what we were doing and failed to further elicit requirements. We should have spent a lot more time in this phase to ensure what we were building was exactly what the client wanted.

We could have spent a lot more time on design. When first given the requirements, we did not know where to start. We explored some ideas, created some small prototypes, and made some design decisions, but we did not create an overarching design for the system as a whole. Most design decisions were up to the developers when it came time to implement the decision. We ended up refactoring a lot of the code multiple times because we were not on the same page. Then when it came time to integrate the microservices we created, there was no good way to do it without major refactoring.

After this semester, there are plans to pass this project on to another team, which makes this the only school project we have worked on that really aimed for maintainability. It required a different style of coding, where readability, documentation, testing, and design played an important role. In many projects, these activities should take more time than the actual coding of the project. In this project, we tried to spend a large amount of time on them, but ultimately spent a lot more time on coding than on the rest. It was still a great learning experience because we were forced to get into the habit of writing tests, documentation, and following designs more than in previous projects and that instilled some good habits into us, even though to make the project a lot more stable and maintainable, we could have greatly improved in this area.

State of the Project

The project is currently functional with some known bugs, and a little less functionality than we planned. We currently have Tyr-CLI and Skadi working and available on NPM (Node Package Manager). Anyone can download it and run it to generate a project. Endor, Koma, and Yggdrasil are also deployed on Iowa State's servers and are also available to the general public. Our project can generate a project, set up a deployment pipeline and monitoring tools. Once the project is deployed, Skadi runs on the project, sending HTTP data, operating system data, and heartbeats to Koma who stores them in Firebase. Then the user can view the monitored data and the health of their application through Yggdrasil.

We have a couple known bugs throughout the application. While the user is creating a project on Endor, they have to use a project name that is available on Heroku and follows the format that Heroku requires. If they don't, the dockerfile will contain incorrect information and will not be able to deploy to Heroku. While using Tyr on Endor, we rearranged the functions so that the user did not have to wait a couple minutes for their project to be created and to receive their files. Tyr was not designed to run in this order, and creates the dockerfile before the name can be validated on Heroku. Because of this, it will take a little redesigning of Tyr and Endor to fix this bug.

Another known bug on tyr is also with creating a project. We have been using a GitHub wrapper to push files to github, because it was simpler to implement than using the official GitHub library. However, it is picky on the special characters allowed in the user's GitHub password and will not allow the project to be pushed to GitHub if they have certain special characters.

Currently, we have an invite and email system that are not quite ready to use, but have been mostly implemented. These will hopefully be some of the first new features added as a lot of the functionality already exists. To replace this system, we have a simple add-a-user feature to the projects that just adds the user without asking them.

When a user signs up for the application, we do not verify their email. They could then use someone else's email or use a fake email. We will need to add a way to verify that it is their email.

Our UI is not currently mobile friendly and scales strangely on different screen sizes. This bug is not noticeable if the user has a large enough screen and does not do much resizing of the screen, but should be addressed in later development of the project.

There were also a couple security features that were overlooked due to our inexperience. The main one brought to our attention was rate limiting on the front end and the API. This should be one of the first things added to the application to prevent a DDOS attack.

Each of these applications can be extended to include more features and more options for the data. Right now, there is one path for everything, but there could be a lot more added to give the user more choice in the set up of their application. We have laid the groundwork for a more complex project generator, DevOps, and monitoring platform.

Future Work

Our goal this semester was to lay the groundwork for this project to be continued after this semester. During our time developing this project, we have brainstormed some work that could extend the functionality which we have built or completely add new features to the product.

More Tool Support

One of the most obvious spots that could be worked on for this project is adding more tools to support. We currently only support one tool per category of tooling type that we support. Adding more tools to support would be a great value add to the project. Below is a list of tools that should be added in the future:

- GitLab
- CircleCI
- Linting Frameworks
- Deployment pipeline for any server in the world
- Mongoose/MongoDB
- Jasmine/Should

Project Management Suite

Project Management is an important concept for projects. Also, DevOps and Project Management are closely aligned. A project management tool which integrate well to a DevOps platform would have a significant impact on our user's productivity. A project management suite with the following features would add value to the project:

- Ability to create, delete, update issues
- Ability to create, update, remove descriptions and acceptance criteria for an issue
- Ability to assign team members to issues
- Ability to label issues
- Ability to track sprints and make issues for sprints
- Ability to view a sprint board
- Ability to update status of the issue (TODO, in progress, done)
- Ability to link issues and commits together
- Ability to link pull requests and issues together
- Ability to view test results and reports per issue
- Ability to view code coverage (diff coverage) for that issue
- Ability to view code quality information for that issues

- Ability to automatically stub tests based on acceptance criteria
- Create reports for issues to determine how likely it is to become a bug in the future

Microservices Framework

Creating code for Microservices is hard and there is a lot of overhead to get Microservices to communicate together. Our team suggests that a code generator is created which can help create API endpoints and client code.

This framework would be defined in a pseudo-code like language that could be parsed programmatically, essentially creating a domain specific programming language for creating microservices and their endpoints. This framework would make it easy to automatically generate APIs and their corresponding documentation. It would also make it easy to provide client code for those APIs. It would generate the client code as a library, so it could be imported into any project where those are needed. Also, it would stub the tests for the API endpoints. Finally, it would write all of the needed to validate the API calls to ensure the proper requests were made and automatically respond with the proper errors.

The goal of this framework would be to decrease the overhead needed to create Microservices.

More Data Monitoring

Currently, we have a couple sections of data that we are monitoring, but this area could be increased. We want to make sure the application is healthy, and running and identify any possible errors that may have occurred. The following are examples that could be added to the Data Monitoring side:

- Ability to identify if an application has gone down and send out an email to the project owners.
- Store errors and error messages to identify frequent errors and bugs
- Record response time by URL

Appendices

Appendix I: Operation Manual

This appendix provides instructions for setting up, testing, and running the Hammer-IO system. It is composed of five distinct subsystems: Tyr, Endor, Koma, Skadi, and Yggdrasil. They are presented in that order because it makes the most sense to test and run them in that order. For example, Endor is dependent upon Tyr, so it is outlined after Tyr. Likewise, Yggdrasil is dependent on the other four subsystems, so it is outlined last.

The instructions here present a subset of the development documentation written for each system. It does not include, for example, the deployment documentation. The complete, up-to-date documentation for each system can be found in its respective README.md file and supporting documents in each code repository. The links to each system’s README are provided for reference.

Tyr

For complete and up-to-date instructions, please refer to the documentation at https://github.com/hammer-io/tyr/blob/master/README.md.

Setup

Prerequisites

Before you can use Tyr, you need to make sure you've done the following:

1. Create a GitHub account (https://github.com/). At this current stage of development, GitHub is the default version control platform for storing and managing your code.
2. Ensure that you linked your TravisCI account to your GitHub account.
3. Create a Heroku account (https://signup.heroku.com/). At this current stage of development, Heroku is the default web hosting service.
4. After creating a Heroku account, visit the following link to find your API key: https://dashboard.heroku.com/account. Make sure to copy it, as you'll need it to sign in to Heroku.

Installation

```
npm install --global tyr-cli
```

CLI Usage

Tyr can be used from the command line or as an imported module. The command line usage is described as follows:

```
tyr [OPTIONS]
```

Options:

- `-V, --version` output the version number
- `--config <file>` configure project from configuration file (see more below)
- `--logfile <file>` the filepath that logs will be written to
- `-h, --help` output usage information

Configuration File (.tyrfile)

Project Configurations

Name	Required	Note
`projectName`	Yes	Must be a valid directory name and cannot be a directory that already exists.
`description`	Yes	
`version`	No	Must match `(number)(.number)*`
`author`	No	For multiple authors, use comma-separated values
`license`	No	

Tooling Choices

Name	Required	Description	Valid Choices
`ci`	Yes	The Continuous Integration tool you want to use	`<None>`, `TravisCI`
`containerization`	Yes	The Containerization tool you want to use	`<None>`, `Docker`
`deployment`	Yes	The deployment tool you want to use	`<None>`, `Heroku`
`sourceControl`	Yes	The source control tool you want to use	`<None>`, `GitHub`
`web`	Yes	The web framework you want to use	`<None>`, `ExpressJS`
`test`	Yes	The test framework you want to use	`<None>`, `Mocha`
`orm`	Yes	The Object-relational Mapping framework you want to use	`<None>`, `Sequelize`

- If Source Control Choice is `<None>`, then CI Choice, Containerization Choice, and Deployment Choice must also be `<None>`.
- If CI Choice is `<None>`, then Containerization Choice and Deployment Choice must also be `<None>`.
- If Containerization Choice is `<None>`, then Deployment Choice must also be `<None>`.

File Format

The configuration file should have the `.tyrfile` extension. Its contents should be in JSON format and should contain the following:

```
{
  projectConfigurations:
      {
      projectName: '{project name}',
      description: '{project description}',
      version: '{version number}',
      author: ['author1', 'author2', ...],
      license: '{license}'
      },
  toolingConfigurations:
      {
      sourceControl: '{source control choice}',
      ci: '{ci choice}',
      containerization: '{containerization choice}',
      deployment: '{deployment choice}',
      web: '{web framework choice}',
      test: '{test framework choice}',
      orm: '{orm framework choice}'
      }
}
```

Endor

For complete and up-to-date instructions, please refer to the documentation at https://github.com/hammer-io/endor/blob/master/README.md.

Installation

Installation for Development

1. `git clone https://github.com/username/endor`
2. `npm install`
3. Setup the configuration file
   - Duplicate `config/default-example.json` into a new file `config/default.json`
   - Fill in any necessary information (either create new accounts or ask an owner)
   - For development and testing, create an Ethereal account to mock the email service. Fill in the email section of the configuration file with this information.
4. Generate the documentation html files
   - `apidoc -i src/ -o docs/`
   - NOTE: You must first have apidoc installed. `npm install apidoc -g`
5. Setup the database
   - Run `npm run createDB && npm run initDB` to create the database and initialize the tables within it.
6. You're all set!

Usage

`apidoc -i src/ -o docs/`	Generate the documentation html
`npm start`	Starts the API server on localhost:3000
`npm test`	Runs the test suite
`npm run lint`	Runs the linter

Koma

For complete and up-to-date instructions, please refer to the documentation at https://github.com/hammer-io/koma/blob/master/README.md.

Setup

1. Run `git clone https://github.com/hammer-io/koma` to clone the repository
2. Run `cd koma`, then `npm install`
3. Setup your Firebase database
   - Create an account at https://firebase.google.com/
   - Create a Firebase project with Realtime Database
   - In the Realtime Database panel, add a new key-value pair `"Test": "Test"`. You can remove this later after the database is populated with some actual data. If you don't add some initial data, the database won't be saved and you'll have to create another new one.
   - Edit the Realtime Database rules, replacing with the contents of `firebase-rules.json`
   - In the `Authentication -> Sign-in Method` panel, enable the Email/Password provider and configure any authorized domains
4. Update the configuration file for development
   - Configuration files are located in the `config/` folder
   - Copy `default-example.json` file to `default.json`.
   - Replace `firebase.databaseUrl` with the URL to your Firebase database.
   - Replace `firebase.serviceAccount` with the Service Account which is downloaded in Firebase underneath `Project Settings -> Service Accounts -> Firebase Admin SDK -> Generate New Private Key`
5. Generate the documentation html files
   - `apidoc -i src/ -o docs/`
   - NOTE: You must first have apidoc installed. `npm install apidoc -g`
6. Initialize your MySql database by running `npm run initTestDB`

Usage

`npm start`	Starts the web server
`npm test`	Runs the unit tests
`npm run lint`	Runs the linter

Skadi

For complete and up-to-date instructions, please refer to the documentation at https://github.com/hammer-io/skadi/blob/master/README.md.

Setup

Create a `.skadiconfig.json` file in the directory where you are launching your application from.

```
{
    "interval": "<optional interval in milliseconds>",
    "apiKey": "<apiKey from koma>",
    "heartbeatUrl": "<server url to koma heartbeats>",
    "osDataUrl": "<server url to koma os data>",
    "httpDataUrl": "<server url to koma http data>"
}
```

Usage

```
const skadi = require('skadi')
```

Heartbeat

With `heartbeatUrl` in the `.skadiconfig.json` file, add the following snippet after your imports.

```
skadi.heartbeat();
```

OS Data

With `osDataUrl` in the `.skadiconfig.json` file, add the following snippet after your imports.

```
skadi.osdata();
```

HTTP Data

To capture incoming requests, add the following snippet before your routes.

```
app.use((req, res, next) => {
  skadi.captureRequestData(req);
  next();
});
```

To capture outgoing responses, add the following snippet after your routes.

```
app.use((req, res, next) => {
  skadi.captureResponseData(req, res);
});
```

Yggdrasil

For complete and up-to-date instructions, please refer to the documentation at https://github.com/hammer-io/yggdrasil/blob/master/README.md.

Setup

Clone the project onto your computer and install dependencies:

```
git clone https://github.com/hammer-io/yggdrasil.git
npm install
```

Before running, make sure you've done the following:

- Install, configure, and start Endor
- Install, configure, and start Koma
- Configure the application
    - `cp config/default-example.json config/development.json`
    - Fill in the development config. For third-party client IDs, ask the project owners or create your own accounts for each.
    - Firebase configs
        - The Firebase instance should be the same one created for Koma (see instructions for creating a new Firebase instance below in the Koma section of this Appendix)
        - On the project overview page, click "Add Firebase to your web app"
        - Copy the configs, convert it to JSON, and put it in the `config/development.json` file
        - When testing for development, you need to make sure to register a new user through the application sign-up process. This will authenticate the user with firebase. None of the test users (e.g. jreach) are setup with firebase, and the application will not work correctly for them.

Usage

`npm start`	Starts the development web server
`npm run lint`	Runs the linter

Appendix II: Alternative Designs

We split our project into subsystems almost from the start because the desired functionality was nicely compartmentalized. Additionally, we reasoned that decoupling the systems would enable us to work independently.

Skadi needed to be a separate library so it can be imported into a project and used as a middleware and service. The only one we would have been able to connect it to would have been Tyr since it is also a library, but the functionalities are disjoint, so it was necessary to keep them in separate microservices.

The two microservices we considered combining were Endor and Koma. We could have sent all the project data to Endor and then stored it in a database or Firebase, and retrieved it via Endor. But there were a few problems with this. We wanted the data to be updated in real time so the user knows if something goes wrong as they are watching it and it would have taken considerable effort to do that through Endor. Also, Skadi is available to the public, so it makes sense that they should also be able to access the backend that stores the data. They would not need all the functionality of Endor to do this, so we put Koma in a separate microservice.

We considered storing all the project's monitoring data in a database. However, with Firebase, real-time updates were quick and simple, making it easy to add heartbeats and immediate monitoring. This could also potentially be a large amount of data if a lot of projects were monitored by our platform. It was easier to handle the large amount of data through firebase than to create our own large database. Although, on upside to the database, is there is a limit in the amount of data and requests we can store and make for free on the database. We will just have to monitor the amount of data that is stored there and make sure to not exceed the free limit.

We considered using Jenkins instead of TravisCI. Both are free to use, but Jenkins would have taken a larger amount of time to get up and running with and requires a dedicated server to run on, while TravisCI is ready to be hooked up to GitHub. We also considered here that our main use would be students and small teams, and TravisCI was very simple and accessible to students. They are able to use it for free, and it takes little knowledge of CI or configuration to use.

Appendix III: Other Considerations

Naming Conventions

We named our subsystems after various things in Norse mythology. In Norse, Yggdrasil is the mythical tree that connects the nine worlds. For us, it connects the microservices of our users. Then we have Koma, which is Norse for *to come/gather/arrive*, since it gathers monitoring information about the users' projects. Tyr is the God of War and brother to Thor, who wields a hammer, which is what we chose for our team name. Yes, the connection is flimsy, but there you have it! Finally, there's Skadi, the Goddess of Winter and the Hunt. She wielded a bow and arrows—the connection here being the arrow-like messages fired at Koma from Skadi.

Endor is the anomaly to this convention. When we first started building Yggdrasil, the frontend and backend web servers coexisted in the same repository. They were each named for moons in the Star Wars universe: Coruscant and Endor, respectively. Heated debate surrounded this departure from the Norse naming conventions, but after a while one side gave up and we were left with two Star Wars moons. Once the frontend and backend were separated, the frontend simply remained "Yggdrasil" and the backend kept its original monicker of Endor. Thus, Endor lives on in infamy...

Appendix IV: Links to Code and Other Resources

Deployed Instances

Yggdrasil	http://hammer-io-test.ece.iastate.edu/
Endor	http://api-hammer-io-test.ece.iastate.edu/
Koma	http://koma-hammer-io-test.ece.iastate.edu/
Team Website	http://sdmay18-19.sd.ece.iastate.edu/ https://hammer-io.github.io/

Links by Subsystem

Tyr

Code Repository	https://github.com/hammer-io/tyr
Continuous Integration	https://travis-ci.org/hammer-io/tyr
Published NPM Package	https://www.npmjs.com/package/tyr-cli
Gitter (Support Chatroom)	https://gitter.im/hammer-io1
Code Coverage Report	https://codecov.io/gh/hammer-io/tyr

Endor

Code Repository	https://github.com/hammer-io/endor
Continuous Integration	https://travis-ci.org/hammer-io/endor
Deployed Instance	http://api-hammer-io-test.ece.iastate.edu/
Code Coverage Report	https://codecov.io/gh/hammer-io/endor

Yggdrasil

Code Repository	https://github.com/hammer-io/yggdrasil
Continuous Integration	https://travis-ci.org/hammer-io/yggdrasil
Deployed Instance	http://hammer-io-test.ece.iastate.edu/

Koma

Code Repository	https://github.com/hammer-io/koma
Continuous Integration	https://travis-ci.org/hammer-io/koma
Deployed Instance	http://koma-hammer-io-test.ece.iastate.edu/
Code Coverage Report	https://codecov.io/gh/hammer-io/koma

Skadi

Code Repository	https://github.com/hammer-io/skadi
Continuous Integration	https://travis-ci.org/hammer-io/skadi
Published NPM Package	https://www.npmjs.com/package/skadi-hammerio

Team Website

Code Repository	https://github.com/hammer-io/hammer-io.github.io
Deployed Instances	http://sdmay18-19.sd.ece.iastate.edu/
	https://hammer-io.github.io/

Appendix V: Acronyms and Definitions

API Application Programming Interface. A well-defined interface through which a user or system can interact with the application.

API key A secret token given to a user authorizing him or her to access specific application functionality through the API.

BDD Behavior-Driven Development.

CI Continuous Integration. The automation of certain aspects of the software development lifecycle, such as testing and merging new code into the existing code base.

CD Continuous Deployment. The automation of application deployment upon specified criteria being met. Closely tied to Continuous Integration.

CLI Command Line Interface. The means by which a user interacts with an application through a command shell.

DevOps A software engineering practice that aims at unifying software development (Dev) and software operation (Ops).[2]

Docker A container platform used for deploying distributed software applications.

Endpoint A web service URI that provides a specific set of functionality.

GUI Graphical User Interface. The visual interface through which a user interacts with an application.

IDE Integrated Development Environment. Software used to develop, test, and run application code.

MS Microservice. An independently-deployable subsystem that interacts with other microservices through a well-defined interface in support of a larger system's functionality.

Node.Js A framework for running standalone JavaScript applications.

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

npm Node.js Package Manager. A tool used to intelligently manage dependencies for a Node.js application. Also refers to the website where Node.js packages are published and hosted.

PR Pull Request. A code source control process by which new code is presented for integration into the existing code base. Pull Requests usually involve the code being tested and reviewed before being approved and merged into the master branch.

TDD Test-Driven Development.

UI User Interface. The means by which a user interacts with an application. Often refers to the frontend components of a web application.