Hammer-io IoT Extension

Final Report

Team: sddec19-24

Members:

Matt Bechtel
Yussef Saleh
Brett Wilhelm
Chakib Ahlouche
Ahmed Sobi

Email: sddec1924@gmail.com
Website: sddec19-24.sd.ece.iastate.edu

Table of Contents

Introduction

Project Statement

The problem that has prompted this project is the need to extend the existing Hammer-io system to support the deployment of software to our client's IoT devices. Some caveats come with this problem though, such as the restriction that the IoT deployment can only rely on the fact that the IoT device is running some sort of Linux. With this requirement comes the request that we support a wide range of IoT devices. Thus we cannot expect to be able to leverage the power of virtualization technologies, such as Docker, as they take up too much of the device's resources.

Goal

The goal of the project is to provide an extension to the current DevOps system, Hammer-io, that will allow clients to deploy their services to the IoT device of their choosing. As of now, the system only allows deployment via Heroku, so we will have to deliver an extension to Hammer-io that allows for the deployment of software to IoT devices. This extension will be in the form of the new services, the 'IoT Deployment Service' and the 'IoT Deployment Manager'. These new services will add on to the existing microservices architecture of the Hammer-io system, and once integrated, should allow for clients to use Hammer-io to push and deploy services to their desired IoT device. The two new services will work in conjunction to provide this new functionality, with the 'IoT Deployment Service' running alongside the rest of the Hammer-io system in a microservices architecture, and the 'IoT Deployment Manager' running on the actual IoT device.

The **deliverables** will be as mentioned above, the extended Hammer-io system with the new services 'IoT Deployment Service' and 'IoT Deployment Manager' added to provide the needed IoT deployment functionality to the client.

Term Definitions

See the references at the end for official documentation of given technologies

- API
  - Application Programming Interface
- Docker
  - A virtualization technology
- Express
  - lightweight web framework for NodeJS that provides easy set up and extensibility
- Git
  - Version control software
- Hammer-io
  - Existing senior design project that we are extending
- Heroku
  - An enterprise deployment service that is used by the existing Hammer-io system
- IoT Device
  - 'Internet of Things' Device, this term is fairly broad and usually refers to a low weight device (such as a raspberry pi) that is connected to the internet. In this Project Plan we will narrow this term a bit defining it as a low resource device that runs Linux, and in particular does not have enough resources to perform virtualization with a technology such as Docker.
- Linux
  - Operating system
- Microservices
  - A type of software architecture that involves making logical splits between components such that each component is loosely coupled with the others in the system. Typically each component (service) in the system is stateless, relying on a data store to house information pertinent to system, this allows for each component (service) in the system to be scaled as needed. The existing Hammer-io system is built using a microservices architecture, our new components will simply add to it.
- NodeGit
  - Library for NodeJS that provides an API to Git commands
- NodeJS
  - Javascript for the backend
- Repository
  - In this paper when we mention repository, we mean a Git repository.

Design

Requirements For the IoT Extension (IoT Deployment Service and Manager)

Functional Requirements

- Extension adds IoT Deployment functionality to the Hammer-io system.
- User can deploy software from their repository on their IoT device using the Hammer-io system.
- The implementation should utilize the existing user access functionality provided by Endor to authenticate deployment requests, making sure that only users who own a given repository or device can perform deployments with them.
- Deployment functionality should be available via the UI or via a CI route using Git Hooks.
- UI should be extended to provide the user easy access to all IoT deployment functionality.
- API endpoints should require a valid user token to perform any action.

Non-Functional Requirements

- API should be developer friendly with documentation for easy use (swagger docs).
- API should respond with correct error codes and messages.
- Hammer-io system should be able to be easily distributed via source code for each service and via docker images with an accompanying Docker Compose script (this allows for the standing up of the cloud services via a single command and a mounted configuration file).

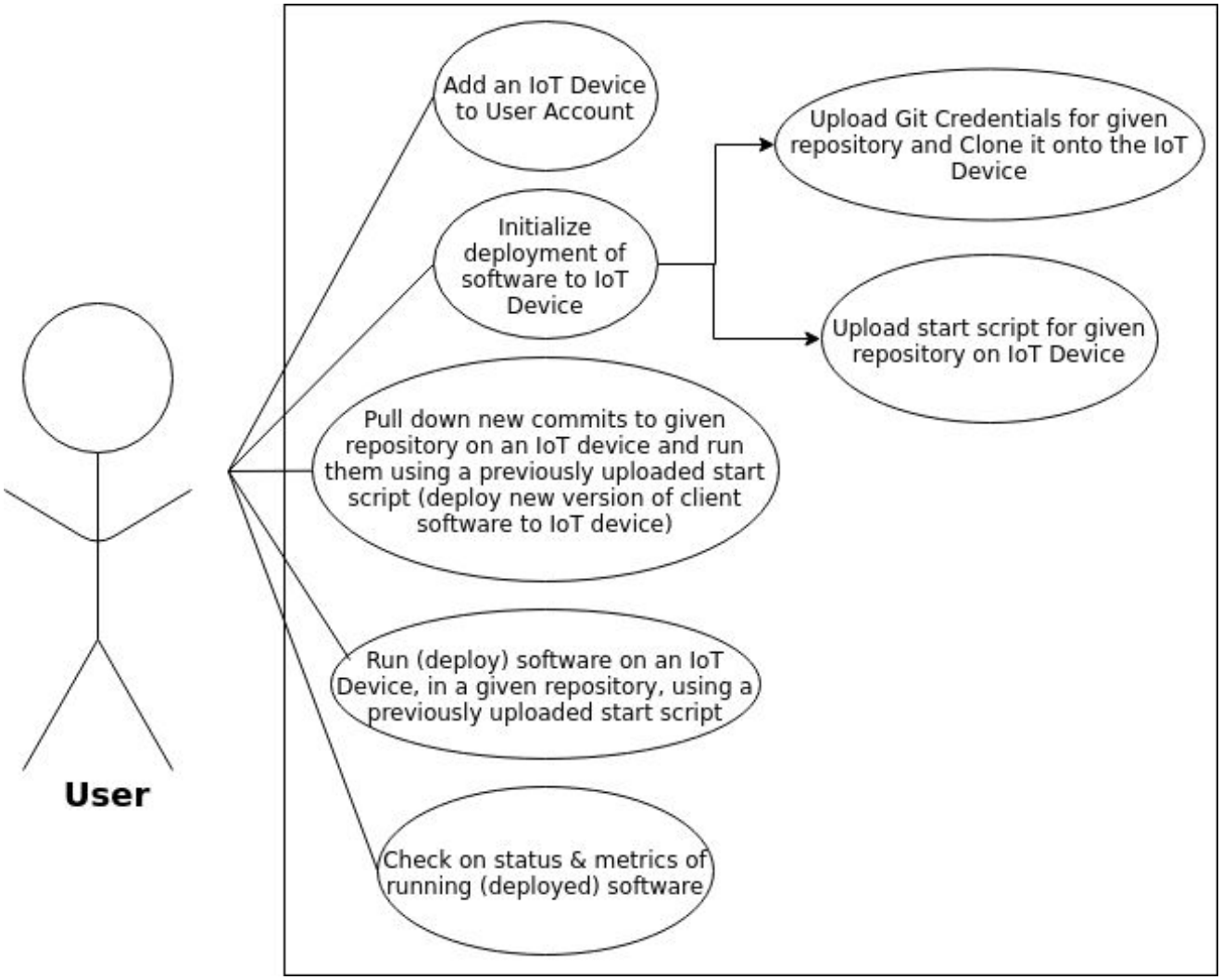
Engineering Assumptions/Constraints

- IoT Device
    - The IoT device is a Linux device that is connected to the internet at an address that is addressable by the IoT Deployment Service.
    - The IoT device is owned by the user and has the IoT Deployment Manager actively running on it.
    - The IoT device does not have enough resources to perform virtualization, thus the usage of Docker is unavailable.

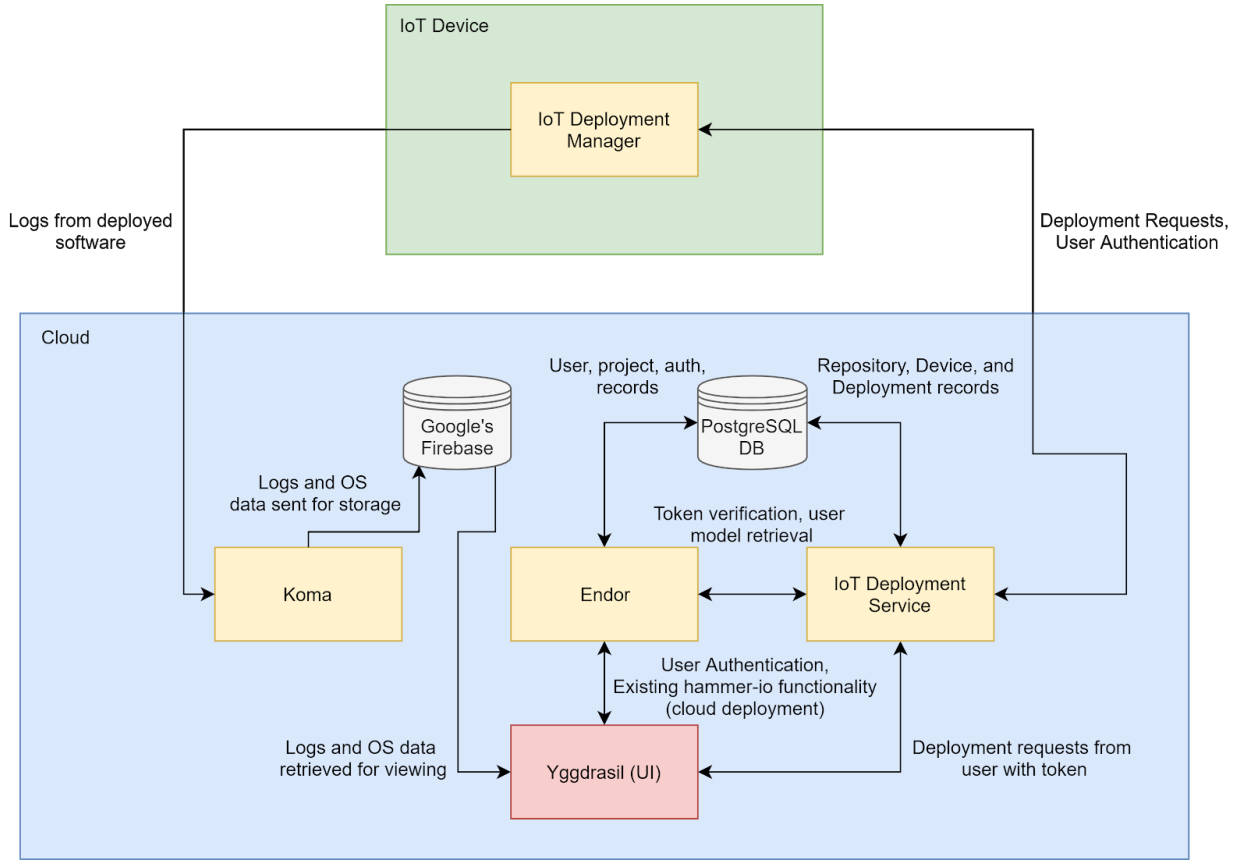
Design Overview

The Hammer-io system with our extension is comprised of 5 services: IoT Deployment Service, IoT Deployment Manager, Endor, Koma, and Yggdrasil. The following diagrams and descriptions should give a good overview of the functionality and components of the system.

Use Case Diagram For the Extension



Component Diagram



Component Definitions

IoT Deployment Manager ***New***

The Deployment Manager will run on the user's device and receive deployment requests directly from the associated Deployment Service. The manager will authenticate all requests with the Deployment Service, ensuring that the token in the request belongs to the user who owns the device they are attempting to deploy to. Once authenticated, the manager will execute the user-provided start script within the repository to start the software. Once running, logs produced by the running software (the deployment) will be sent to Koma for aggregation.

IoT Deployment Service ***New***

This is a new service adding functionality for IoT deployment. This service will be hit directly by the user to perform deployments to a given IoT device. It will authenticate all inbound requests with the user access endpoints provided in Endor. The service will communicate with IoT Deployment Manager running on the user's IoT device to perform deployments of the user's software. The service will be responsible for storing deployment records, repository information, and device information.

Yggdrasil

User Interface. Surfacing all of the system's functionality in usable manner. Yggdrasil was built by the previous team.

Endor

Existing backend web service built to handle all user management and authentication. Also handles existing cloud deployment requests, which involves the storing of a user's credentials and direct contact with cloud services such as Heroku to facilitate deployments.

Koma

Data aggregation service for all deployed instances. Collects log and OS data.

Implementation Details

Technical Stack

Programming Languages

- JavaScript was used to implement all of the services in the Hammer-io system

Frameworks

- Backend
    - The NodeJS Express framework was used for all of our backend services, both in the original Hammer-io system and the extension.
- Frontend
    - ReactJS was used for the creation of the UI

Database

- Relational Database
    - Dialect
        - PostgreSQL
    - ORM
        - Sequelize was used to facilitate all DB connections (NodeJS ORM for SQL)
- NOSQL Database
    - Google's Firebase
        - We utilize Firebase for 'real-time' access to particular data (in particular for logs from running deployments and os data)
        - Firebase also allows for the UI to directly connect to the datastore instead of having to hit one of our backend services to perform the query

Other utility libraries

- Nodegit
    - The Nodegit library is what we used within the IoT Deployment Manager to interface with the client's git repository.
- Request
    - The Request package allows us to easily communicate across services, this is what is used in the IoT Deployment Service to proxy a given deployment request to a user's device.

Extension Implementation Definitions

IoT Deployment Service

Backend web server written using NodeJS and the Express framework. This service will run alongside the existing Hammer-io system in the cloud and will serve as the deployment 'hub' for all IoT Deployments. The service will manage information about each user's repositories, devices, and deployments in a PostgreSQL database with the Sequelize ORM. A deployment will be performed via a combination of a repository and devices. This means that a user will perform an IoT deployment by 1) adding a repository they wish to deploy 2) adding a device they wish to deploy to 3) making a deployment request as a function of those two things. The actual running of the software, the actual deployment, will be done by the IoT Deployment Manager which will be running on the client's IoT device, so the service will simply verify a user owns a given set of devices then proxy those requests to the requested devices. So, for example, a user could add a repository and a bunch of devices, they would be able to, with one request, deploy that repository to all of those devices by simply sending a valid token, a repository ID, and a list of device IDs to the IoT Deployment Service.

IoT Deployment Manager

The IoT Deployment Manager was also built using NodeJS and the Express framework. The manager runs on the client's IoT devices and facilitates the actual deployment functionality. To do this, without virtualization such as Docker, the manager uses the Nodegit library to interface directly with the clients remote Git repository. The manager receives requests from the IoT Deployment Service to perform deployments. These requests contain the repository credentials as well as the script provided by the user in order to start the software in the repository. So a deployment on the manager side will go a bit like this, it receives a request to perform a deployment it then 1) verifies ownership of the device and the validity of the token with the IoT Deployment Service 2) uses the credentials in the request to clone the user's Git repository 3) runs the contents of the start script with the new repository as the working directory 4) manages the software's processes in order to allow for the stopping of a deployment via a request or failure 5) responds to the IoT Deployment Service with success or error message.

Implementation Issues and Challenges

Packaging Issues

We were planning to containerize 4 of our systems (Yggdrasil, Endor, Koma and IoT-Deployment-Service) in order to make the Hammer-io system easy to distribute and use. We focused on the packaging of the system, specifically in regard to the distribution of Docker images, for each of our services. We wrote a Docker Compose script that was supposed to allow users to stand up the cloud portion of the system (Yggdrasil, Endor, Koma, IoT-Deployment-Service). This script was also to pull down the new docker images from Docker Hub, mount the user's configuration, and then start the services on a Docker network. After writing the yml files we faced a lot of problems. The system didn't respond and connect on the hub and we didn't have time to focus on fixing these issues as we had other priorities.

UI Issues

As the previous development group had created a web-based GUI for interfacing with the system, we had originally decided to extend that UI to include support for the functionality that we were adding. Unfortunately, this addition was postponed near the end of our development time to be continued by a future development group working on Hammer-io. Due to a combination of poor time-management and some personal issues the UI team was going through at the time, progress was slower than it should have been and ultimately the extension was shelved in favor of finalizing the completable aspects of the project. As it does seem that development of Hammer-io and the extension will continue, this will likely be one of the first things to be completed.

Deployment Logging Issues

Our original plan was to collect the right logs in the data aggregator Koma from the deployed software, but during the integration there were many challenges, such as: the inability to know exactly what data to collect from the deployment manager, not a very in-depth understanding of the data aggregator code done by our predecessors, ambiguous data packaging and grouping in Koma which resulted in a hard time constructing good and understandable tables in Firebase data storage. Although documentation of the data aggregator was good, it was not enough to use Koma accordingly, and to integrate the Service/Manager piece to it.

Open Issues

*** These will be passed off to the next team ***

UI

The IoT Deployment UI was not completed this semester but still needs to be implemented for the system to be in a usable state (no user really just wants to make API calls directly). This was planned to be an extension of the existing UI Yggdrasil.

Packaging

The full packaging of the cloud portion of the Hammer-io system (Endor, Koma, Yggdrasil, IoT Deployment Service) still needs to be done to allow for easy environment stand up for the client.

Deployment Logging

As of now the deployments made by the Service and the Manager do not collect logs and store them for the client. We got a bit of a start on this but were not able to complete it. The next team should implement this to finish the IoT Deployment suite of Hammer-io.

Further cloud integration (AWS)

Our client mentioned that this system should also integrate with other cloud providers beyond Heroku, specifically AWS. We did not have time to implement this functionality so this will be left to the next team.

Testing

Functional Testing

Manual Testing

Much manual testing was done to verify the functionality of our extension, because of the nature of it. Such a thing was a bit hard to test using automated testing methods, especially for the integration portion of it. We performed automated regression testing on the existing system (tests developed by the previous team) to make sure that our newly added pieces to Endor, Koma, and Yggdrasil do not break any functionality, but beyond that we went through the use cases of our extension manually to verify that the functionality of the system is what is expected. This manual testing involved standing up the Hammer-io system, with our extensions active, and performing a deployment onto an IoT device. Of course for convenience sake, this IoT device was going to be a Docker container running Raspbian so that the dev team may have an easily reproducible IoT environment to test with.

Automated Testing

Automated testing for the original system was done leveraging Mocha and its unit testing functionality. Such tests were ran automatically before the new code is pushed to production to verify that the code will function as expected once deployed. We did not have time to implement unit tests/automated testing for the extension.

Regression Testing

Regression Testing was done every time a new build is created to make sure that changes to the code do not break the existing functionality. This testing was done by executing all unit tests and verifying that they pass.

Non-functional Testing

Much of the non-functional testing was done manually, as it is difficult to measure many of these qualities with any sort of tool. For example, ensuring ease of use is not really something that we can test with a piece of software.

Looking specifically at 'IoT Deployment Service', it was fairly simple to manually ensure that our API is easily extensible and easy to utilize. Supportability requirements are similar in this aspect, as it was fairly telling us that our systems do not work on the necessary systems should our testing environment stop working properly. In establishing that the service meets our standards for reliability, we researched load balancing techniques. In general these load balancing techniques are external to the deliverables, but as long as the system is stateless and data driven it is easily load balanced (the IoT Deployment Service is both of those things).

Looking at 'IoT Deployment Manager', we see only a few differences. With regard to reliability, we built endpoints on the manager to allow the user, and the IoT Deployment Service, to 'ping' their device. Finally for security testing we simply played the role of a malicious user trying to make unauthenticated requests.

Testing Standards

- Unit Tests
    - Each commit pushed to the remote repository should contain unit tests that correspond to the functionality added in that commit
    - The author of the new commit is also responsible for producing unit tests for its functionality. In order to pass code review, these tests must be verified by the reviewer to cover every line of the new code (100% code coverage).
- Integration Tests
    - For each new component we will require that there be tests written to verify its functionality in the system as a whole. These were important markers for how the system will function once deployed.
- System and Acceptance Testing
    - Manual testing by developers was 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 was the standard for Acceptance Testing. These take place on a weekly basis at the team meeting.
- Code Coverage
    - To verify that our unit tests cover every line of code, we leveraged the built in Intellij code coverage tool.

Testing Results

The kind of testing that we performed, manual, unit, integration, and regression led to a multitude of results. Specifically, manual testing told us how the system looks from a user perspective, allowing us to verify non-functional requirements such as ease of use. The unit tests gave us insight into which pieces of functionality were working and when at a local scale, allowing us to see if we had broken a given method's functionality with new commits. Integration testing was a bit difficult as it could only be performed at the end of the semester manually when the extension was nearly complete (this also functioned as acceptance testing). We showed the client our system functionality as a whole and received their approval. Regression testing resulted in knowledge about the past and present of the system, showing us whether a new addition to any service had broken previous functionality. This is similar to unit testing and was provided by the previous team.

Operation Manual

All repositories reside under the group in Git Lab https://git.ece.iastate.edu/sddec19-24

IoT Deployment Service

First, clone the repository and step into the iot-deployment-service directory.

For the IoT Deployment Service to function properly, other parts of the system must be running and active. These dependencies include a Postgres database (on port 5432), Endor, and Koma. Also to perform any deployments, the IoT Deployment Manager must be active wherever the client wishes to deploy their software. Once the system dependencies are met, one will have to install all of the dependent packages of the service by running the command '**npm install**', if node or npm is not installed, please install it first otherwise this command will not work. The '**npm install**' command will read the 'package.json' file within the project and download the dependent packages.

Next, we'll need to configure the database connection by placing a JSON configuration file within the ./config directory in the service. This file will contain the database url and credentials and will allow the Sequelize ORM to initialize the connection and manage the data. Here is the basic format of the required JSON file:

```
{
 "development": {
  "username": "root",
  "password": null,
  "database": "database_development",
  "host": "127.0.0.1",
  "dialect": "postgresql"
 },
 "test": {
  "username": "root",
  "password": null,
  "database": "database_test",
  "host": "127.0.0.1",
  "dialect": "postgresql"
 },
 "production": {
  "username": "root",
  "password": null,
  "database": "database_production",
  "host": "127.0.0.1",
```

```
  "dialect": "postgresql"
 }
}.
```
For more information about the above format see the Sequelize documentation https://github.com/sequelize/cli/blob/master/docs/README.md

To start the system once the dependencies are met and the DB configuration is set, one must set some environment variables in the context to configure the service. These environment variables and their options include:

- **LOG_MODE**
  - This variable can be set to 3 modes depending on the context the user would like to use the system in. 'info' for logging all information statements, 'error' for only logging error statements, and 'debug' for logging everything.

- **ENDOR_URL**
  - This variable must be set in order for the IoT Deployment Service to communicate with Endor. Endor will be used for all user management, so when a request comes into the deployment service it will use this address to call off and verify the user's credentials.

Once the above environment variables are set to the desired values, the IoT Deployment Service can be started by running the command '**npm start**'. If the service successfully connected to the database you should see some SQL in the log output (this is Sequelize setting up the models and making sure the ORM is initialized). If any database connection issues occur, make sure the database is running and you have created the database configuration file for Sequelize.

On successful startup of the service, you will have access to the following API:

- Device
    - **Create device**
        - Method
            - POST
        - Url
            - **/device**
        - Body
            - Device object
                - deviceName
                - ipAddress
                - Active
    - **Update device**
        - Method
            - PUT
        - Url
            - **/device**
        - Body
            - Device object with id
                - id
                - deviceName
                - ipAddress
                - Active
    - **Delete devices**
        - Method
            - DELETE
        - Url
            - **/device**
        - Body (search criteria)
    - **Search devices**
        - Method
            - GET
        - Url
            - **/device**
        - Body (search criteria)
    - **Ping**
        - Method
            - GET
        - Url
            - **/device/ping**
        - Body (search criteria)

- Repository
    - **Create Repository**
        - Method
            - POST
        - Url
            - **/repository/**
        - Body
            - Repository object
                - repositoryName
                - gitHttpAddress
                - gitUsername
                - gitPassword
    - **Search repositories**
        - Method
            - GET
        - Url
            - **/repository**
        - Body (search criteria)
    - **Update repository**
        - Method
            - PUT
        - Url
            - **/repository/**
        - Body
            - Updated repository object with id
                - repositoryName
                - gitHttpAddress
                - gitUsername
                - gitPassword

- Deployment
    - **Start deployments**
        - Method
            - POST
        - Url
            - **/deployment**
        - Body
            - repositoryId
            - deviceIds (list)
    - **Stop deployments**
        - Method
            - POST
        - Url
            - **/deployment/stop**
        - Body
            - deploymentIds (list)
    - **Search deployments**
        - Method
            - GET
        - Url
            - **/deployment**
        - Body (search criteria)

*** In the above API search criteria refers to whatever fields of that object the user is looking for, for example, on a device, search criteria could include “active”:true which would then result in the returning of all device records for the given user where the field “active” is set to true. ***

*** All requests in the above API require a valid token in the header which can be gathered by registering an account with Endor and then logging in. The IoT Deployment Service authenticates all requests against Endor. With that, the above API is user specific, that means each route will only allow the user to access records/things that they ‘own’ or have previously added to the system ***

IoT Deployment Manager

First, clone the repository and step into the iot-deployment-manager directory.

For the IoT Deployment Manager to function, the instance of the IoT Deployment Service that the manager is associated with must be active and reachable. The address for the IoT Deployment Service must also be provided in the environment. To use the IoT Deployment manager, you will need to install NodeJS on the device you wish to deploy to and then you will run the command '**npm install**' to download all of the system's dependencies. Once the associated IoT Deployment service is active and reachable, and the manager has all of dependencies set, some environment variables need to be set in the context in order to configure the system. These environment variables and their options include:

- **LOG_MODE**
    - This variable can be set to 3 modes depending on the context the user would like to use the system in. 'info' for logging all information statements, 'error' for only logging error statements, and 'debug' for logging everything.

- **DEPLOYMENT_SERVICE_URL**
    - This should be set to the url of the deployment service to which the device that the manager is running on is associated. This allows the manager to verify device ownership (authenticate incoming requests).

With the environment variables set, the command '**npm start**' should be ran to start the manager. Once the manager is active on the device, it should be able to be registered with the IoT Deployment Service as a 'device'. Upon successful registration of the device, one should then be able to perform deployments by making requests to the IoT Deployment Service API, which will then proxy those requests to the correct devices, running an instance of the IoT Deployment Manager.

The following API that is provided by the IoT Deployment Manager will be used directly by the IoT Deployment Service:

- Deployment Manager
    - **Start deployment**
        - Method
            - POST
        - Url
            - **/deploymentManager/deploy**
        - Body
            - Hydrated deployment record (this is generated by the IoT Deployment Service when a deployment is requested)
                - Repository record
                - Device record
                - Deployment record
    - **Stop deployments**
        - Method
            - POST
        - Url
            - **/deploymentManager/stop**
        - Body
            - deploymentIds (list)
    - **Get active deployments**
        - Method
            - GET
        - Url
            - **/deploymentManager/status**
    - **Ping**
        - Method
            - GET
        - Url
            - **/deploymentManager/ping**

*** All requests in the above API require a valid token in the header which can be gathered by registering an account with Endor and then logging in. The IoT Deployment Manager authenticates all requests against the IoT Deployment service to make sure that 1) the provided token is valid 2) the owner of the token also owns the device their deploying to. ***

Happy Path IoT Deployment Scenario

- PostgreSQL DB, Endor, Koma, IoT Deployment Service are all active
- IoT Deployment Manager is active on the device they wish to deploy to
- User creates a hammer-io account with Endor
- User logs in with the new account's credentials to receive a token. This token will be used to authenticate subsequent requests.
- The user adds the repository they wish to deploy to their account by hitting the 'Create Repository' endpoint with the following information in the body
    - gitUsername
    - gitPassword
    - gitHttpAddress
    - repositoryName
    - startScript
- The user adds a device that has an instance of the IoT Deployment manager running on it. This is done by hitting the 'Create Device' POST endpoint on the IoT Deployment Service with the following information about the device in the body:
    - deviceName
    - ipAddress
- Once both a device and repository are added to the system, a deployment may take place as a function of those two things. To perform a deployment the user should hit the 'Start Deployments' POST endpoint on the IoT Deployment Service with the following information in the body:
    - repositoryId
    - deviceIds (list)
- The above request, when successful, will search the db for the repository and device ids provided, and will proxy a deployment request to each device with the correct repository information (to each IoT Deployment Manager).
- The software then started deployment request will be active on whatever device was deployed to (will be actively managed by an instance of the IoT Deployment Manager).
- All deployments can be managed via the IoT Deployment Service API.

Manual for original parts of the Hammer-io System

*** What follows was made by the previous team. The original can be found here: https://sdmay18-19.sd.ece.iastate.edu/docs/Final_Report_Spring_2018.pdf in Appendix I***

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(re
  q, 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

References

Docker: https://docs.docker.com/
Github: https://developer.github.com/v4/
Hammer-io: http://sdmay18-19.sd.ece.iastate.edu/docs/
Hammer-io Design Document: https://hammer-io.github.io/docs/Design_Document_v2.pdf
Heroku: https://devcenter.heroku.com/categories/platform-api/
Mocha: https://mochajs.org/api/mocha/
NodeGit: https://www.nodegit.org/api/
NodeJS: https://nodejs.org/en/docs/
NPM: https://docs.npmjs.com/
Previous Team: https://sdmay18-19.sd.ece.iastate.edu/docs/Final_Report_Spring_2018.pdf
Sequelize: https://sequelize.org/v5/