

The Design and Implementation of CollabTest, a Collaborative Testing Framework

Joshua Chorlton (3035 350 038)
Mingxuan Du (3035 339 292)
Richard Wong (3035 350 648)

ELEC 2603
Vincent Tam
December 6, 2016

Table of Contents

Introduction	2
Overview of Project	2
Build Process	2
Design Decisions	2
Server	2
Database	3
User Interface	4
Messaging and I/O	5
Logical Flow	5
Login	5
Authentication	6
Project Creation	7
Adding Test Cases	7
Running Test Cases	7
Performance Concerns	8
Test Plan	8
Basic Test Cases	8
Login and User Sessions	8
Project Creation	9
Test Case Submission	9
Test Suite Execution	9
Advanced Test Cases	10
Support for Multiple Languages	10
Test Suites with Multiple Test Cases	10
Link Sharing	12
User Manual	13
Assumptions	13
Primary Interactions	13
Authentication	13
Project Ownership	13
Uploading Test Cases	13
Running a Project Test Suite	13
External Libraries	14
Golang Packages	14
NPM (Javascript) Packages	15

Introduction

Overview of Project

CollabTest is a collaborative testing platform which allows users to compile and execute small programs on shared test suites, which users can extend. Users log in using their GitHub account, and are able to upload files representing test input and expected output on a per-project basis.

Test cases are run in isolated execution environments in Docker containers equipped with only bare minimum dependencies. The container images used currently support the compilation and execution of standalone C, C++ and Java programs.

Build Process

The project's primary dependency is that the Docker engine is installed on the server host. Docker containers are used to run the Golang web server and build the Javascript front-end. This limits the requirement of manual installation of dependencies on the host server, and enables deployment on any platform that supports running containerized applications.

The next most important dependency is git, which is needed to clone the source code onto the host server. The build process is started using the GNU make utility. The CollabTest container image can either be built directly on the server, or pulled from Docker Hub. Other make targets build the Javascript front-end, start up a PostgreSQL database container, and create the container network.

Once the dependencies are installed, the source code can be cloned from github.com/jchorl/collabtest (assuming the user has repository clone permissions). Inside the `collabtest` directory, the user can build the project and start a server on the local host by running `make dev`. When complete, CollabTest will serve requests at <https://localhost:4443>.

Running the Application

Running the application is simple, and the only requirement is to have Docker installed. Simply install Docker (<https://docs.docker.com/engine/installation/>), `cd` to the project directory, and type `make dev`.

Design Decisions

Server

The server was written using Golang because it is fast and has great support for concurrency. Golang is also a compiled language, which make deployment very easy as only a single executable must be deployed. Compared to Java, Golang is very similar in execution

speed and developer productivity, but Golang uses significantly less memory. An HTTP server running using Golang will use 5 MB of memory while a JVM will use minimum 50 MB. Compared to Python, Golang is statically typed which reduces the prevalence of errors in a large codebase. Golang is also many times faster than Python.

The HTTP server framework used by CollabTest is called [labstack/echo](#), a framework built on the standard HTTP library of Golang. Internally, the Golang HTTP library uses a worker model built on top of lightweight threads. Golang creates new threads for each client request. Labstack/echo extends upon this with better routing, middlewares, and request parsing. The API routing of labstack/echo uses a radix tree for fast lookups. We have middlewares for injecting a database connection and a Docker client connection into request contexts so each request can communicate with the database and Docker engine. Request data is parsed from JSON using labstack/echo's methods that wrap Golang's JSON deserializer.

The production server consists of two Docker containers: an API container running the Golang application, and a PostgreSQL container for the database. A Docker network is created between the containers so our API serving container can communicate with our PostgreSQL container.

Database

The database schema consists of three tables listed below. In addition to the default indices on the primary keys for each table, we also added indices on foreign keys to make finding relationships faster.

```
CREATE TABLE users (  
    id SERIAL PRIMARY KEY,  
    github_id INTEGER,  
    created_at TIMESTAMP NOT NULL,  
    updated_at TIMESTAMP NOT NULL,  
    deleted_at TIMESTAMP DEFAULT NULL  
);  
  
CREATE TABLE projects (  
    hash VARCHAR(8) PRIMARY KEY,  
    user_id INTEGER REFERENCES users (id),  
    name VARCHAR(31) NOT NULL,  
    created_at TIMESTAMP NOT NULL,  
    updated_at TIMESTAMP NOT NULL,  
    deleted_at TIMESTAMP DEFAULT NULL  
);
```

```

CREATE TABLE runs (
  id SERIAL PRIMARY KEY,
  project_hash VARCHAR(8) REFERENCES projects (hash),
  stdout TEXT NOT NULL,
  stderr TEXT NOT NULL,
  created_at TIMESTAMP NOT NULL,
  updated_at TIMESTAMP NOT NULL,
  deleted_at TIMESTAMP DEFAULT NULL
);

CREATE INDEX project_hash_idx ON runs (project_hash);
CREATE INDEX github_id_idx ON users (github_id);

```

When choosing the database, we considered two options: MySQL and PostgreSQL. Everyone on the team had used both in production, and their features were very similar. The deciding factor was familiarity, and PostgreSQL was the database that we were more comfortable with. We did not consider MS SQLServer because it was not available on Linux Docker containers.

We did not use a NoSQL datastore such as MongoDB/HBase because we knew the layout of our schema and did not need the benefits of a schemaless datastore. The flexibilities of a relational SQL database make some future extensions easier such as analytics which rely on JOINS. Scalability advantages with NoSQL database definitely would be useful if we wanted high availability, but we do not expect to outgrow the capabilities of PostgreSQL anytime in the near future.

User Interface

Most web applications use Javascript to request data and display dynamic content, as Javascript enables a feature-rich web application experience. There are many common modern Javascript frameworks that make the design and implementation of feature-rich user interfaces far easier. Some common frameworks include ReactJS, AngularJS, and Vue.js. We opted to use ReactJS because it encourages clean and modularized code. ReactJS allows a developer to define components and build a user interface from modular and nested components.

Each component can have data passed to it (using props) and can store state (using state). For example, we created a component to handle adding new test cases. That component receives the project hash passed in as a prop, so that on submission, the component can tell the server which project to add the test cases to. The component also maintains state on which files have already been submitted. A test case is comprised of 2 files, and it is only submitted when both files are present. When a user adds the first file, the state is updated. When the user adds the second file, the state is checked to verify that both

files are present and then the test case is submitted. Furthermore, the parent component need only have a component in it such as `<SubmitTestCase hash={ hash_value } />`, which keeps the parent code tidy and encapsulates all test case submission logic and presentation into its own component.

Additionally, ReactJS is smart about choosing when to re-render components. For example, components will be re-rendered when their props change. When a user selects a different project, the components that display project details will have updated props indicating a different selected project id. React will re-render those components so that they display information on the newly selected project instead of stale information on the previously selected project.

The user interface also uses Redux, which allows the web-application to store global state in the browser and make it accessible to all components. This is very powerful because it eliminates the need to pass all data through components. For example, many components depend on project details. Instead of passing all project details through many levels of ancestry that do not need those details, individual components can query the global state for whatever information they need. This helps to keep components focused on their own purpose without needing to pass through information that is unrelated to themselves.

The user interface is built using Webpack, which packages up all Javascript and CSS into one minified and optimized Javascript file. Building complex user interfaces with modern frameworks often requires many Javascript files that import each other and many CSS files to define styling for different components. While this is great from a code organization standpoint, there are complications with loading many external resources on a web page. For instance, some resources should only be loaded after their dependencies. Additionally, it is important from a development perspective to store code in a human-readable format, but this often hinders performance because human-readable Javascript and CSS files are larger than they need to be. Webpack solves these problems by optimizing, minifying and deduplicating all Javascript and CSS and storing the result in a single Javascript file.

Messaging and I/O

There are a few major components that must successfully communicate with each other for CollabTest to function correctly. Foremost, the Go server must be able to communicate with the PostgreSQL database to allow for storage and retrieval of data. Communication between the Go server and PostgreSQL database is done using HTTP network requests. Furthermore, the Javascript web application must communicate with the Go server in order to send requests and fetch data. Most communication between the web application and the server is done using JSON via HTTP calls. All communication between the client and server is done via TCP/IP, and encrypted using TLS. The Go server exposes a REST API for intuitive operations, such as creating projects. The main exception to using JSON and REST APIs is file uploads, which are all done using HTTP form-data. Form-data is

the current web standard for managing file uploads. The server also must communicate with the Docker engine in order to run test cases in containers. Docker provides a Go client library for communicating with the Docker engine, which uses HTTP calls under the hood to issue commands to the Docker engine. Files are also an important aspect of CollabTest, as each test case is saved as an input and expected output file on disk. File I/O is handled using Go's `io` package, which makes operations like `stat`, `ls`, and `cp` very easy.

Logical Flow

Login

Each project is tied to a user who serves as a project owner. The owner ultimately has authority over the project and can tweak project settings, delete test cases, and even delete the project. A user must therefore login before they can begin creating or modifying projects. GitHub was selected as an ideal method of authentication because CollabTest is designed for programmers and the vast majority of programmers have GitHub accounts. Signing in with GitHub is done via OAuth2. When a user begins the flow by selecting to Login with GitHub in CollabTest, the user is redirected to GitHub to provide permissions to CollabTest. Upon successful granting of privileges to CollabTest, GitHub redirects the user to a specific URL hosted by CollabTest, specifying a code as a query parameter. CollabTest reads that code and sends it back to GitHub to ensure that the code is in fact a valid code issued by GitHub. This is a standard procedure in OAuth authentication flows because any user on the internet can attempt to call the callback supplied to GitHub with a fake code, but authentication is supposed to connect the user to a legitimate GitHub user. GitHub then confirms that the code is a real code issued by GitHub and also sends some information back, including the user's unique id on GitHub. This unique id is imperative because a user's GitHub account id will never change. The id is stored in the CollabTest database so that if the user logs out and logs back in, the user can be queried by GitHub id and all of their information can be retrieved.

Authentication

Some requests require authentication. For example, creating a project must be done via authenticated request because the project must be tied to a valid user. There are two main ways of making authenticated requests: using headers and using cookies. An example of an authorization header for an authenticated HTTP request is: `Authorization: Bearer <session_id>`. Cookies allow a web application to store data in a client's web-browser that are sent with some HTTP requests. Some requests do not include cookies for security reasons, and the security policy for when to include cookies is defined when the cookie is set. A sample cookie header that might be sent with an authenticated request is: `Cookie: Authorization=<session_id>`. There are various security concerns associated with using headers and cookies. However, there are secure ways of using both headers and cookies for authentication. Ultimately, cookies were selected as they can be configured to be sent with all requests to the server automatically and securely, whereas headers must be stored and added manually to all requests.

The next major design decision was choosing the format with which to store authentication information and send it to the server. The most basic way of storing authentication information might be to simply store the user's id in the cookie, so when the user makes an authenticated request, CollabTest can simply grab the user id from the cookie and know exactly who the user is. The problem with storing the user id in plaintext is that any user can set cookies on their requests, so a malicious user could set `user_id=1` in their request and get access to user 1's data. In order to avoid this possibility, JSON Web Tokens (JWT) are used. A JWT is created using claims (data), a keyed hash function, and a key. There are three components of a JWT: the header, payload and signature. The header includes implementation details such as which hashing algorithm is used. The payload is a base64 encoding of the claims. The signature is the output of a keyed hash function run on the claims using a specified key. The entire JWT is used as the cookie. When a user logs in, the CollabTest server sets a cookie value to a JWT created using the user's id as a claim and a global secret key for the signature. When the user issues a subsequent request to the server, that cookie is included in the request. The JWT can be verified, because all keyed hash functions are verifiable using the secret key. Assuming the user sends a valid JWT, the server can get their user id from the claims of the JWT and know exactly who the user is. JWTs make session management easy, because the server does not need to store session ids and user ids in a database. Instead, that information is stored in a JWT in a cookie in the user's browser.

Project Creation

When a user issues a request to create a project, a random 6-character hash is generated. That hash becomes a unique identifier for the project. Due to the requirements of CollabTest to function, mainly the ability for anybody to visit a project link and run their program against the test cases, it is important that every project has a unique URL. This problem is easily solved by giving each project a unique identifier. However, giving projects incrementing numeric ids can be dangerous, because malicious users could easily upload test cases for projects that they do not own and ruin the experience for all users. To avoid such issues, a random hash is assigned to projects such that the links to projects are not efficiently obtainable.

In addition to generating a random hash, each new project receives its own directory on the production server's filesystem. That directory is used to store the test cases associated with the project. Since the hash of each project must be unique, the project folders take the names of the hashes to ensure that there are no folder naming collisions.

Adding Test Cases

When a user adds a test case, two files must be supplied: the input to supply the program and the expected output of the program. It is a general assumption that programs being tested are deterministic. That is, given the same input any number of times, a program should be expected to output the same result. From this assumption, it does not make sense to save multiple test cases with different outputs. Only one test case must be stored for a

given input. Hashing makes comparing the contents of an input easy. Therefore, when a test case is uploaded, the input file is hashed using MD5 hashing. The input file is then saved in the project's folder under the name `<hash>.in`. Similarly, the output is also saved in the project's folder under the name `<hash>.out`. When a user attempts to upload a test case with a previously used input file, the input file is hashed and the previous test case gets overwritten.

Running Test Cases

Running test cases is one of the more complex functionalities involved with running CollabTest. Thankfully, Docker makes it easy and efficient to provide each test case with an isolated execution environment. The process starts when a user supplies a program file and requests that tests be run against that program. Based on the filename extension, CollabTest will select an environment to handle that type of program. For each test case, a new Docker container is created with the selected environment and the program source code file is copied over. The program is compiled inside the container, as Docker provides the container with the required environment to do so. The test case files are then copied into the Docker container using Docker's Go client. The program is run with stdin redirected from the test case's input file, and the output is read from stdout and returned to the Go server. The Go server then uses a Go diffing library to diff the output with the expected output, and returns the diff to the user. If the test case ran successfully, diffing the actual and expected output will indicate that there is no difference.

Performance Concerns

Since Go is a generally efficient language with inbuilt concurrency support and low memory overhead, the webserver should be very capable of handling high volumes of load. Similarly, the application does not run any complex queries on the database, and the database schema is quite simple. All common queries are optimized with indexes as well. Therefore, the database should not be a performance bottleneck either. The main concern for performance is executing user code. In general, executing user-submitted code creates many complications including security and performance. Thankfully, Docker helps solve security concerns by running each user-submitted application in its own isolated container. Users can therefore only affect their own container, and not steal data from the server or other containers. However, a malicious user could upload a performance-intensive application, which could starve the webserver of resources because both the web server and running user programs share the same host. Docker provides the ability to pass ulimit limits to containers, and so all containers that run user code are limited in CPU usage and memory usage. This should ensure that a single user cannot starve the entire server. Another issue is high volumes of load, where many users would be running many containers at the same time. Even if those containers are fairly basic, this could still impact performance. Thankfully, we expect that user programs will be quite simple and should not be very resource intensive. If this becomes an issue, the CPU and memory limits can be tweaked to ensure that the

server always has the necessary resources. If this became popular, user code should probably be run on a separate cluster such that it cannot impact the server itself.

Design Changes

Throughout the development process, some critical design decisions had to be changed. Initially, the project did not have any concept of users. Any anonymous user could create a project and share the link with their friends. This is ideal because the barrier of entry is low and usage is very easy. However, there are many valid reasons for every project to have an owner. Thinking ahead, it is possible that a project owner may want to delete a project. Additionally, if projects become configurable, an owner may want to change a build command for a project. A project owner may decide to delete an invalid test case uploaded by a random user. All of these actions should not be publically accessible. For this reason, we introduced the concept of users and project owners. Projects became tied to a Github ID, and only that Github user has administrative authority over the project.

Furthermore, when the project began, the frontend web application was an afterthought. All group members were primarily interested in server-side development, which largely took priority. When it was absolutely essential to create a frontend to make the application usable, a very basic frontend was created using simple HTML and vanilla Javascript. However, as the process grew more complex, there was a need for a complicated web application to support all of our desired functionalities. Due to these requirements, we introduced React and Redux to the project, which made building a complex and rich web-application far easier. While adding these Javascript libraries introduced immense complication, particularly with build process and loading resources, these dependencies also made the display of information and extendability drastically easier.

Future Design Improvements

We made decisions in several areas which would allow our group to produce a working prototype in the limited time allowed for the project, but would need to change for a full-scale, publicly accessible application. Some of these decisions involved our server architecture, feature set, and flexibility of execution.

In our prototype, we use a single machine to run our web server along with our tests. This works fine for a few users at a time, but would not be able to handle spikes in usage. Ideally, the web server would run separately from a compute cluster which would handle test case execution, and work could be distributed among a wider hardware base.

We restricted the supported languages to only C, C++, and Java. This reduced the overall complexity of managing multiple configuration files and environments for different programming languages. Support for more programming languages would be one of the first improvements made if we wanted to further develop the application.

Finally, for simplicity of test case execution, we made the assumption that all test programs would only receive input from the standard input stream. This may be sufficient for most simple programs, but there could be use cases where input may come from other sources, like the file system. Perhaps a program is provided a filename and operates on the

contents of an existing file. Our platform would experience greater adoption if it can fulfil more of the needs of our expected user base.

Test Plan

The application must be running in order to execute the tests. Refer to [Running the Application](#) to get the application running.

It is assumed that these tests are executed in the order that they are listed as some of them require that the previous test was successful. This also makes it easier for the tester to avoid many redundant steps.

Basic Test Cases

Login and User Sessions

- 1) Open a new tab and navigate to <https://localhost:4443> (accept the self-signed cert if necessary).
- 2) Click "Login with GitHub" and give the application the requested permissions on your GitHub account.
- 3) You should be redirected back to CollabTest, but see a project creation screen.
- 4) Create a project by typing in a project name and clicking "Submit". This project will enable us to ensure that your session is indeed kept.
- 5) Open a new tab.
- 6) Navigate to <https://localhost:4443>.
- 7) You should see the project that you created. This means that cookies are properly keeping track of the logged in user.

Project Creation

- 1) Open a new tab and navigate to <https://localhost:4443> (accept the self-signed cert if necessary).
- 2) Login with GitHub if necessary.
- 3) You should see your created project there.
- 4) Refresh the page. The project should still be there. This means that the project was properly persisted to the database and does not only exist in the browser.

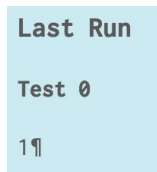
Test Case Submission

- 1) Open a new tab and navigate to <https://localhost:4443> (accept the self-signed cert if necessary).
- 2) Login with GitHub if necessary.
- 3) A created project should already be selected.
- 4) In a file explorer, open the project directory and navigate to projects/square. We will use this simple squaring program to test adding a test case.
- 5) Under square/tests, drag 1.in and drop it over the input box that says "Please drop input file here".

- 6) Under square/expected, drag 1.exp and drop it over the input box that says "Please drop output file here".
- 7) An alert should pop up indicating success, and the test case should appear as "Test 0" on the right.
- 8) Refresh the page and make sure that "Test 0" still appears on the right. This means that the server is aware that there is a test case for this project.
- 9) Click on "Input" and "Output" near where it says "Test 0" and a new tab should open displaying the input/output.

Test Suite Execution

- 1) Open a new tab and navigate to <https://localhost:4443> (accept the self-signed cert if necessary).
- 2) Login with GitHub if necessary.
- 3) A created project should already be selected.
- 4) The selected project should already have the previously added test case.
- 5) In a file explorer, open the project directory and navigate to projects/square.
- 6) Drag square.cpp and drop it over the input that says "Please drop application file here".
- 7) You should see the following Last Run result:



Last Run

Test 0

1

- 8) This means that Test 0 was run and the output was 1, as expected. If the output was anything else, a diff would show the expected versus actual output.
- 9) Drag broken.cpp, also in the square directory, and drop it over the input that says "Please drop application file here".
- 10) You should see the following Last Run result:



Last Run

Test 0

+8

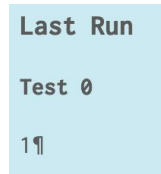
- 11) This is correct because instead of the expected 1 being output, 8 was output, which is hardcoded into broken.cpp

Advanced Test Cases

Support for Multiple Languages

- 1) Open a new tab and navigate to <https://localhost:4443> (accept the self-signed cert if necessary).
- 2) Login with GitHub if necessary.
- 3) A created project should already be selected.

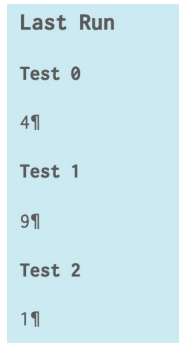
- 4) The selected project should already have the previously added test case.
- 5) In a file explorer, open the project directory and navigate to projects/square.
- 6) Drag Solution.java and drop it over the input that says "Please drop application file here".
- 7) You should see the following Last Run result:



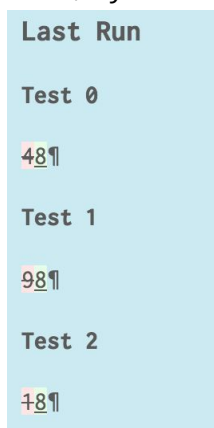
- 8) This means that Test 0 was run and the output was 1, as expected. If the output was anything else, a diff would show the expected versus actual output.
- 9) Now both Java and C++ applications have been tested against the same test suite.

Test Suites with Multiple Test Cases

- 1) Open a new tab and navigate to <https://localhost:4443> (accept the self-signed cert if necessary).
- 2) Login with GitHub if necessary.
- 3) A created project should already be selected. There should already be one test case.
- 4) In a file explorer, open the project directory and navigate to projects/square. We will use this simple squaring program to test adding more test cases.
- 5) Under square/tests, drag 2.in and drop it over the input box that says "Please drop input file here".
- 6) Under square/expected, drag 2.exp and drop it over the input box that says "Please drop output file here".
- 7) An alert should pop up indicating success, and another test case should appear on the right.
- 8) Under square/tests, drag 3.in and drop it over the input box that says "Please drop input file here".
- 9) Under square/expected, drag 3.exp and drop it over the input box that says "Please drop output file here".
- 10) An alert should pop up indicating success, and another test case should appear on the right.
- 11) Refresh the page and make sure that "Test 0", "Test 1", and "Test 2" appear on the right.
- 12) In a file explorer, open the project directory and navigate to projects/square.
- 13) Drag Solution.java and drop it over the input that says "Please drop application file here".
- 14) You should see the following Last Run result:



- 15) This means that all three test cases ran successfully, since the diffs show no differences.
- 16) Furthermore, try running broken.cpp again. The results should appear as follows:



- 17) This obviously indicates that all 3 test cases failed, as expected.

Link Sharing

- 1) Open a new tab and navigate to <https://localhost:4443> (accept the self-signed cert if necessary).
- 2) Login with GitHub if necessary.
- 3) A created project should already be selected.
- 4) Near the project title, there should also be a link similar to the following:

Link: <https://localhost:4443#/projects/cGQNf1PZ>
Created: 11/22/2016, 9:01:44 PM

- 5) Copy that link, and open it in a new incognito window. You should see a similar view to a logged in user, seeing the project title, the test case submission form, and the already submitted test cases. This is what an anonymous user would see by visiting a project link (since you are in incognito mode, CollabTest cannot identify you as a user. You are anonymous.).
- 6) Try running the test cases against the square program again. Behaviour should be the same as previous test cases.

- 7) This means that you can share your project link with your friends, and they can add test cases and run their programs against the test suite without being logged in. Only you can manage the project when you are logged in.

User Manual

Assumptions

This user manual assumes that CollabTest is running on the local host, and the end user can reach the web app at the sample URL `https://localhost:4443`. Details about running the server are written in this report under [Running the Application](#).

If the user runs the app locally, the SSL certificate will be self-signed, and therefore must be explicitly allowed by the browser.

Primary Interactions

Authentication

The first time a user visits CollabTest, they will be brought to a login page, where they can click the button to authenticate themselves. In the absence of a GitHub account, the user can visit <https://github.com> to create an account.

Upon authentication, the user is brought to a landing page containing a listing of projects that they own, and an interface to create new projects. Projects each have a name designated upon creation.

Project Ownership

The creator of the project is considered to be the project owner. The owner of a project has the privilege to delete test cases and delete the project. An automatically generated link is also available, and can be shared with collaborators so that they can contribute their own test cases.

Uploading Test Cases

Once a project is created, test cases can be added. Users can upload pairs of associated input and expected output files to form a test suite. Projects form the natural division of test suites.

Running a Project Test Suite

Users can test a program against the test suite by uploading a source code file (currently, CollabTest supports C and Java programs). Upon testing a program, the program's output is matched against the expected output, and differences are reported to the user. An absence of differences indicates that the user has achieved the expected output for a given test case.

External Libraries

Golang Packages

The following Go packages (or parts of packages) are imported by CollabTest or other libraries that CollabTest imports. Their functionalities range from web server frameworks to database ORMs and even logging frameworks.

"github.com/Microsoft/go-winio",
"github.com/Sirupsen/logrus",
"github.com/dgrijalva/jwt-go",
"github.com/docker/distribution/digest",
"github.com/docker/distribution/reference",
"github.com/docker/docker/api/types",
"github.com/docker/docker/api/types/blkiodev",
"github.com/docker/docker/api/types/container",
"github.com/docker/docker/api/types/filters",
"github.com/docker/docker/api/types/mount",
"github.com/docker/docker/api/types/network",
"github.com/docker/docker/api/types/reference",
"github.com/docker/docker/api/types/registry",
"github.com/docker/docker/api/types/strslice",
"github.com/docker/docker/api/types/swarm",
"github.com/docker/docker/api/types/time",
"github.com/docker/docker/api/types/versions",
"github.com/docker/docker/client",
"github.com/docker/docker/pkg/tlsconfig",
"github.com/docker/go-connections/nat",
"github.com/docker/go-connections/sockets",
"github.com/docker/go-connections/tlsconfig",
"github.com/docker/go-units",
"github.com/jinzhu/gorm",
"github.com/jinzhu/inflection",
"github.com/labstack/echo",
"github.com/labstack/echo/context",
"github.com/labstack/echo/engine",
"github.com/labstack/echo/engine/standard",
"github.com/labstack/echo/log",
"github.com/labstack/echo/middleware",
"github.com/labstack/gommon/bytes",
"github.com/labstack/gommon/color",
"github.com/labstack/gommon/log",
"github.com/labstack/gommon/random",
"github.com/lib/pq",

```
"github.com/lib/pq/oid",  
"github.com/matttn/go-colorable",  
"github.com/matttn/go-isatty",  
"github.com/opencontainers/runc/libcontainer/user",  
"github.com/pkg/errors",  
"github.com/sergi/go-diff/difflib",  
"github.com/valyala/fasttemplate",  
"golang.org/x/net/context",  
"golang.org/x/net/context/ctxhttp",  
"golang.org/x/net/proxy",  
"golang.org/x/sys/unix",  
"golang.org/x/sys/windows",
```

NPM (Javascript) Packages

The following NPM packages are used to run and build the Javascript frontend:

```
"react-scripts": "0.6.1",  
"font-awesome": "^4.6.3",  
"immutable": "^3.8.1",  
"react": "^15.3.2",  
"react-dom": "^15.3.2",  
"react-dropzone": "^3.6.0",  
"react-immutable-proptypes": "^2.1.0",  
"react-redux": "^4.4.5",  
"react-router": "^2.8.1",  
"redux": "^3.6.0",  
"redux-form": "^6.0.5",  
"redux-thunk": "^2.1.0"
```