

6-1-2016

# MEAN Web Application Development with Agile Kanban

Jared Petersen  
*Western Oregon University*

Follow this and additional works at: [https://digitalcommons.wou.edu/honors\\_theses](https://digitalcommons.wou.edu/honors_theses)



Part of the [Computer Sciences Commons](#)

---

## Recommended Citation

Petersen, Jared, "MEAN Web Application Development with Agile Kanban" (2016). *Honors Senior Theses/Projects*. 105.  
[https://digitalcommons.wou.edu/honors\\_theses/105](https://digitalcommons.wou.edu/honors_theses/105)

This Undergraduate Honors Thesis/Project is brought to you for free and open access by the Student Scholarship at Digital Commons@WOU. It has been accepted for inclusion in Honors Senior Theses/Projects by an authorized administrator of Digital Commons@WOU. For more information, please contact [digitalcommons@wou.edu](mailto:digitalcommons@wou.edu).

---

# MEAN Web Application Development with Agile Kanban

By Jared Petersen

An Honors Thesis Submitted in Partial Fulfillment  
of the Requirements for Graduation from the  
Western Oregon University Honors Program

Dr. Scot Morse  
Thesis Advisor

Michael Ellis  
Thesis Advisor

Dr. Gavin Keulks  
Honors Program Director

Western Oregon University  
June 2016

---

# Acknowledgments

Firstly, I would like to thank my advisors, Scot Morse, Michael Ellis, and Gavin Keulks. You have provided me with much help and guidance over the past year.

Finally, I would also like to thank my fiancé, Amy Keithley, for being my rubber duck when it came to debugging parts of the web application. Without her, some of the bugs may have never been fixed.

---

# Table of Contents

<b>Abstract</b>	<b>1</b>
<b>Introduction</b>	<b>2</b>
<b>Getting Started</b>	<b>7</b>
<b>Developing the Backend</b>	<b>14</b>
<b>Developing the Frontend</b>	<b>22</b>
<b>Software Engineering &amp; After Action Review</b>	<b>31</b>
<b>Appendix: Application Images</b>	<b>35</b>
<b>Appendix: Code</b>	<b>46</b>
<b>Bibliography</b>	<b>47</b>

---

# Abstract

I spent one year developing a project management web application in order to gain a better understanding of the software engineering process. The software was built on a technology stack of MongoDB, ExpressJS, AngularJS, and Node.js which is more commonly referred to as the MEAN stack. The experience has exposed me to a new set of tools, software practices, and engineering principles that have left me with a deeper understanding of what it means to be a software engineer and the incredible amount of time and work that is involved in designing and implementing a full-scale software application.

# Introduction

## Project Management Software Market

The majority of the product offerings in the online project management software industry are fairly homogenous and none are particularly excellent. The applications are typically overly complex or are not complex enough to be useful as a dedicated project management application. There is not a happy middle ground and the industry needs to be shaken-up by a new competitor to alleviate this issue.

In general, online project management software takes one of two forms: embellished to-do lists or Kanban. To-do lists are the most common and straightforward way of tracking tasks that need to be completed. Despite this, they are not well suited to managing projects with many moving pieces as they can become overwhelming to users as the project size grows. Over time, a simple to-do list can easily turn into an unscalable wall of tasks, making it difficult to determine the overall status of the project.

Kanban is a more modernized approach. It is a card-based scheduling system that was originally developed by Toyota in the late 1940s to improve the efficiency of their manufacturing facilities. Toyota sought to model their production system on the U.S. supermarkets of the time and developed Kanban as the operating method to facilitate the Just-In-Time (JIT) nature of these stores [1]. More recently, the software industry has adopted and modified the original Kanban system to manage the software development processes [2]. Rather than using Kanban cards to track materials in the manufacturing process, cards are used to represent bug fixes or new product features. The simplest form of this adaptation involves three columns titled “Backlog”, “In-Progress”, and “Complete”. As bug fixes or feature additions are being worked on and eventually completed, the cards associated with the work are moved between the columns to communicate their statuses to the team. This provides a satisfying visual feedback loop to team members and keeps them accountable for the work that they are assigned.

Jira and Trello are two well-known adherents of the Kanban model. While Jira exclusively caters to software engineers, Trello instead uses the model to help people in a variety of professions organize the things that they need to get done. It accomplishes this through infinitely customizable columns that can be tailored to each project's workflow and a slick user interface that makes adding tasks and moving them through the columns a satisfying experience. However, Trello is still not adequate when it comes to managing a large project. It does not have a meaningful team collaboration system and or provide project metrics, which are useful to understanding the overall trajectory of a project and the participation amongst members.

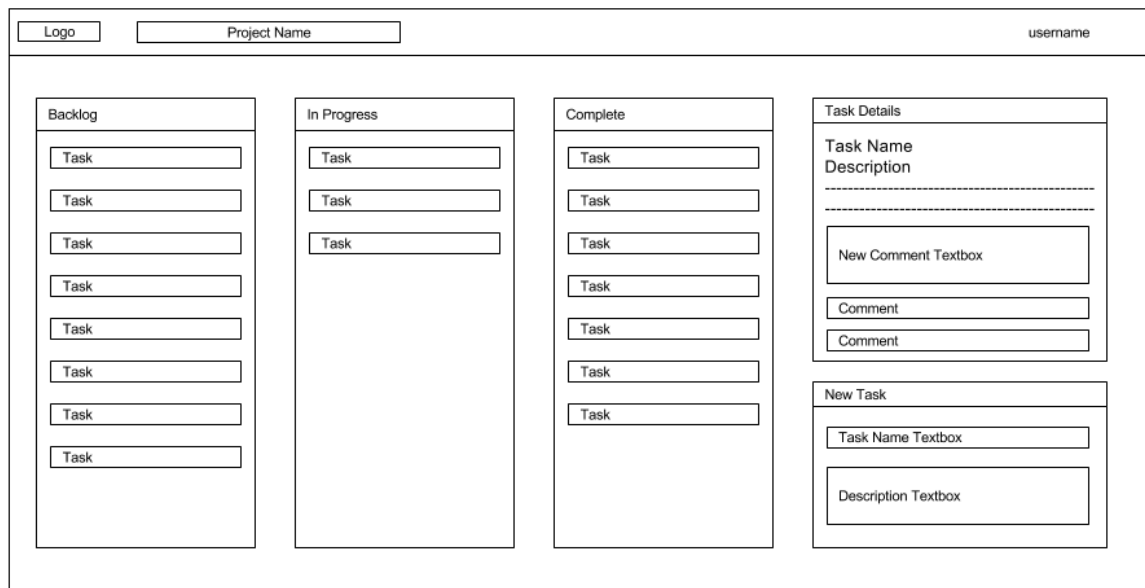
This inadequacy carries over to the rest of the project management software offerings; there is not a single application that possesses the ideal mixture of features that allow teams of any size to collaborate – regardless of their geographic location – while providing team leaders the information that they need to ensure the success of their projects. I recognized this issue and thought that I could resolve it by building my own application. The program would be based on the software industry's adaptation of Kanban and would ideally feature advanced charts, easily accessible project data, a calendar system, and a messaging system, all of which would be built in natively. This application would serve as user's single destination when it came to organizing projects and collaborating with others.

This vision for a new project management application was lofty and I recognized it would be very difficult for me to fully execute on it considering that I am only a single developer with a limited amount of time. Instead, I would focus on creating a partial version of this overall application that had the smallest set of features that was necessary for it to be able to stand on its own as a viable product. The benefit of doing this is that if users did not respond well to the application and it was necessary to completely abandon the project, I would be able to avoid wasting a lot of time building something that no one would want to use. If you are going to fail, fail faster.

## **Application Vision**

One of the core pieces of functionality that I knew would be critical to the minimum viable product is the Kanban board as it is the main point of interaction and organization within the application. Each project would have its own dedicated Kanban board (see

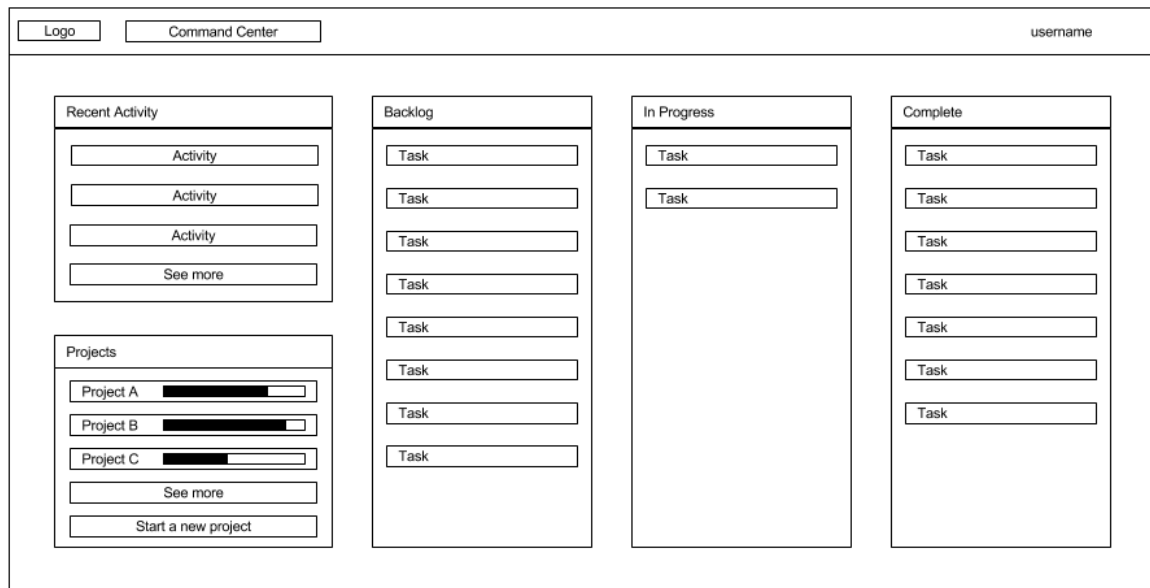
Figure 1.1) and users would be able to add and remove task cards from it as necessary. To indicate that the completion status of a task has changed, users would simply drag and drop the associated task card into the appropriate status column. Clicking on the card title would bring up a more detailed view on what needs to be accomplished for that particular task. In this view, users would be able to edit the task description, set a due date, assign it to one of their team members, and add subtasks to help them break the task down into smaller pieces.



**Figure 1.1** Original wireframe mockup of the project Kanban board

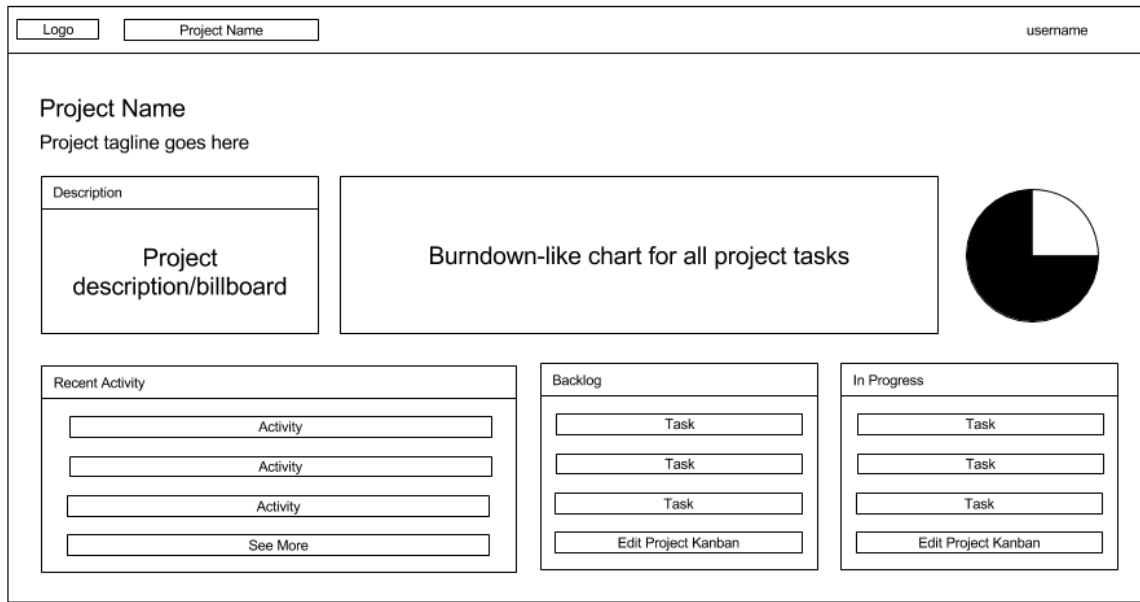
The application would give users their own personal Kanban board as well. These individual boards (see Figure 1.2) would help users manage the tasks that have been assigned to them across all of their projects – a feature that I found has not been widely or well-implemented across the existing project management applications. This board would effectively become a user’s early morning newsfeed that prepares them for the work ahead that day. If a user changes the status or detail information for one of their tasks here, the change would be reflected in its respective project Kanban board immediately. The personal Kanban board would be the “killer app” of the application that I set out to build.





**Figure 1.2** Original wireframe mockup of the individual Kanban board

The individual project dashboard (Figure 1.3) is not necessarily a killer app for the software, but it is important nonetheless. The dashboard would satisfy a team's desire for information about the general status and trajectory of their project through the use of several charts. A single pie chart would be used to display the number of tasks that are currently in each of the Kanban columns and a burndown chart would display the amount of work that the team needs to complete versus the amount of time that the team has left to complete it. Burndown charts are more prominently used in Agile software development teams but other professions can easily take advantage of them for their own projects. I knew that charting this data might be difficult because of how complex software charting libraries are from previous exposure, so I was a little worried that getting this feature to work might take more time than I had to complete the project.



**Figure 1.3** Original wireframe mockup of the individual project dashboard

Features like a calendar system and messaging system were immediately off the table for this reason. They are important to project management and team collaboration and are also a mainstay of project management software but I did not have the time to complete them. Instead, they would need to be added in later after I received some form of validation from users in the application's beta. At that point, I could begin building out a way to monetize the software through a Software as a Service business model, make the application publicly available, and then work on adding those features over time. Breaking down development into these smaller deliverable chunks follows the tenants of Agile software engineering and good business practices in general.

# Getting Started

## Choosing a Stack

One of the first decisions that a software engineer has to make when starting a new project is choosing the language and toolsets to use. It is an incredibly important decision as choosing the wrong language for the job can result in many wasted hours trying to make something work when it could have been done more easily in another language. When I started out, I really wanted to write the application in Django, which is Python-based web framework that was originally created back in 2003 by the web development team at the *Lawrence Journal-World* newspaper [3]. I had seen several recent job postings that requested applicants that were experienced in Django and I had heard several good things about it from colleagues. Additionally, Dr. Morse, my thesis advisor was experienced in Python and I thought that I could consult him whenever I ran into some issues. This is not the way that a programming language should be chosen for a project, but at the time Django seemed like a really good choice.

I started the process of learning Django in late May 2015 through completing several of the official online tutorials but I grew concerned about using the framework during my Summer software engineering internship. I attended one of the company's Lunch and Learn sessions – a program in which a senior engineer informally presents a software engineering topic over lunch. During a session lead by senior software engineer Trevor Moore, I learned more about service-oriented architecture (SOA) which involves building software in terms of components that can stand alone and provide a service to other components via a communication protocol. According to Moore, Django's approach is more monolithic and tightly couples the user interface application layer, the backend data processing layer, and the database together in a manner that makes it very difficult to make architectural changes later on. Applications that use the SOA architecture are a lot more maintainable because the software components are only loosely-coupled and can easily be switched out. All of the logic related to each

component's specific mission is only located in one place rather than being bound to another part of the application that is not closely related.

I spent the previous year working with other students in the Computer Science department to build a monolithic web application in ASP.NET using Entity Framework and did not enjoy the experience. The application was effectively a black box and making even small changes to the database was an undertaking, even with using database migrations. I did not want a repeat experience of this, so I decided to pursue the SOA approach. The application architecture would consist of a backend REST API (REpresentational State Transfer Application Programming Interface) built in Node.js, a frontend user interface built in AngularJS, and a MongoDB database. This is more commonly known as the MEAN stack (MongoDB Express.js AngularJS Node.js), a term first coined by MongoDB developer Valeri Karpov in a blog post in 2013 [4]. Each of these tools are very useful on their own but can become even more powerful when used together.

One of the main benefits of using this software stack is that engineers only have to work with one programming language from the database up [4]. Querying the database is done in JavaScript and all of the data in the database is stored in JavaScript Object Notation (JSON), the backend API is in JavaScript and serves data to consumers in the form of JSON (without needing to convert it), and the entire frontend runs on JavaScript and consumes the JSON data. As Karpov puts it, "By coding with JavaScript throughout, we are able to realize performance gains in both the software itself and in the productivity of our developers. [4]" JavaScript is the de facto language of the web, so it makes sense to use it throughout a web application.

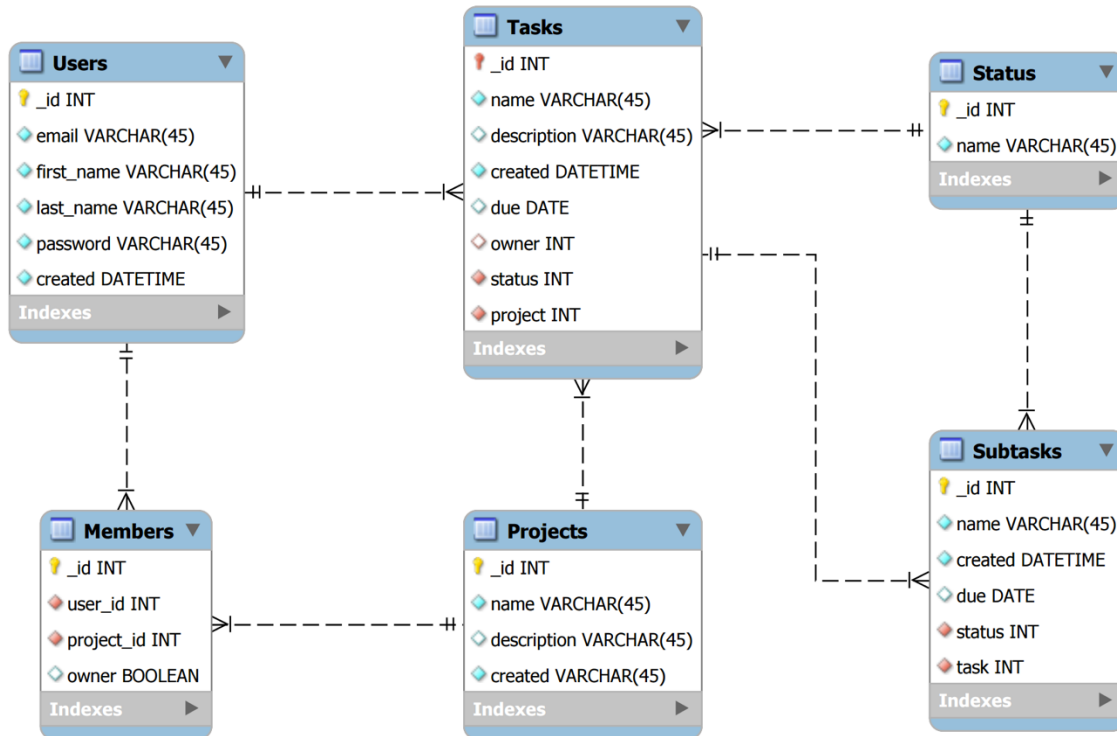
Another sometimes overlooked benefit of the MEAN stack is the large and vibrant development community for each of the individual tools. Learning a stack is a lot easier when there is a large number of tutorials on how to get started and some of the common issues that developers have run into in the past. Each of tools are open source with a friendly open source community as well, so getting support is a simple GitHub issue or Stack Overflow post away. Django is also open source but the development community is a lot smaller, so finding solutions to some of the early problems that I experienced with

the framework while going through the official tutorials was a lot more difficult and made switching over to the MEAN stack much more desirable.

## **Adventures in Database Design**

My first development user story was to get a prototype MongoDB database up and running on my local computer. This would allow me to begin storing the project and user data that comprises the application. MongoDB is a NoSQL database and completely removes the traditional table structure and data normalization methods, instead storing data in JSON through collection and documents. These structures can be loosely compared to the relational database table and records, respectively, as documents store a single instance of data and collections store documents [5]. I found it difficult to translate my experience with relational database schema design to modeling the data in MongoDB's system, which was exasperated by MongoDB's methodology of connecting collections together (similar to a SQL join).

As part of the modeling process, I determined that the project management application has four main entity types that can be considered tables in a relational database: Users, Projects, Tasks, and Subtasks. Each User can own zero or more Projects and Tasks, each Project can have one or more User members and can have zero or more Tasks, and each Task can have zero or more Subtasks (See Figure 2.1). This is simple in a relational database because most of the relationships are one-to-many. The exception here is the relationship between Users and Projects which can be broken down into a bridge table that stores the relationships between the two and a Boolean indicator for a project member's status as the project owner.



**Figure 2.1** Entity-Relationship (ER) diagram, created in MySQL Workbench

MongoDB manages relationships differently than relational databases. It offers the traditional one-to-many and one-to-one relationships but does not offer a normalized way to model many-to-many relationships. This is because the database does not support joining collections like a traditional relational database, instead forcing users to manually join collections together in what is known as an application-level-joins [6], [7], [8]. It instead offers two techniques for linking collections together: document embedding and document linking [8] [9]. Document embedding involves simply including the data into the document directly. In the oft-used example of managing a blog database, blog posts would be a collection and comments would be an array attribute of the blog post that contains a series of comment objects:

```

{
  "title": "My First Post",
  "body": "Hello and welcome to my first blog . . .",
  "created": "2016-04-17T07:46:27.152Z",
  "comments":
  [

```

```

    {
      "text": "Work on your grammar . . .",
      "user": "ellism",
      "created": "2016-04-18T08:30.21Z"
    },
    {
      "text": "A good first try . . .",
      "user": "keulksg",
      "created": "2016-04-19T00:10.01Z"
    }
  ]
}

```

This approach is not optimal if the embedded documents need to be queried extensively on an individual basis because MongoDB does not support that functionality [7].

Document linking is a suitable alternative that is similar to foreign keys used in relational databases. Instead of embedding the necessary document directly, the document can be referenced via a unique ID and later joined together manually in through the previously mentioned application-level-join [7], [8]. In the continuing the blog post example, the blog post document would store an array of comment document references that uniquely identify specific comments. To retrieve all of the blog post and comment data, the querying application would need to make a database request for the blog post and then make another database request for all of the comments associated with the blog post based on the comment document references that were retrieved from the original database request. Document embedding and linking are inherently logical methods of facilitating one-to-one and one-to-many relationships but implementing many-to-many relationships requires a different way of thinking.

In order to be successful with MongoDB, developers have to get out of the joining and database normalization mindset and think entirely in terms of what would make sense if they were organizing a set of JSON files. I struggled with this concept in my first attempt at using MongoDB and switched over to using Postgres temporarily because of the issues that I encountered with modeling many-to-many relationships. The difficulty involved led me believe that MongoDB was not the best tool for this application. This bothered

me, as others before had successfully managed to use the database with their relational data and still touted its superiority. I gave up my previous database modeling technique of drawing out Entity Relationship (ER) diagrams and experimented with modeling the data in JSON format, which completely shifted my perspective on the problem. Rather than thinking in terms of database normalization and connecting tables together, I was able to think about how JSON objects could be linked together.

The solution that I arrived at uses a technique that I later learned is referred to as “one-way-embedding” [6]. To handle the many-to-many relationship between Users and Projects, I created a Project schema that references the User schema via a “members” array attribute stores multiple User document references and an “owner” attribute (representing leadership over a project) that stores a single User document reference. The following is an example JSON document that uses this collection schema:

```
{
  "_id": "56c8c0d3b910597464140927",
  "name": "Project Management Web Application",
  "description": "Tool to organize projects",
  "created": "2016-03-25T04:29:27.152Z",
  "owner": "55ebab212f8851540ea0397f",
  "members":
  [
    "55ebab212f8851540ea0397f",
    "55ebab262f8851540ea03980"
  ]
}
```

The other approach that I could have taken to model the one-to-many relationship between Projects and Users is “two-way-embedding” in which the User collection also has a reference to the projects that a user is a member of and the projects that they own via arrays of Project document references [6]. Since the application focuses on projects first and then on the users that are part of the project, it was not necessary to be able to query from both sides. When a user is authenticated, they are essentially given their ID which is used to query the projects that they own or are a member of. There is not a use



case that justifies the added logic that would need to be implemented to ensure that references in each collection are accurate and up-to-date.

# Developing the Backend

## REST API Background & Authentication System Design

With the database completed, I was able to move on to developing the REST API. This is the crux of the web application because it serves as the single public access point for manipulating data in the database. REST APIs are structured based on resources (nouns) and the HTTP methods used to act on those resources (verbs). Resources describe a particular data item like projects and depending on the HTTP method used, the API will deliver back different data (in a machine-readable format like JSON or XML) or perform different actions related to the resource. In continuing the previously mentioned blog application example, a user might request the API to create a new blog post with a POST request, retrieve a post's information with a GET request, update a particular post with a PUT request, or delete a post from the database with a DELETE request all via the same API URL endpoint like `https://api.yourapplicationhere.com/posts` (also denoted as `/posts`) for example. Data parameters can be passed in the URL and in the body of the request so that the API can retrieve the data the user is requesting or create a new data item entirely.

The REST API approach is a good fit in this case as it locks all of the logic of the application in one place and serves the necessary data in an easily consumable format to whomever or whatever requests it. This fulfills the requirements of the previously determined micro services architecture and allows the UI layer of the program to be very light and multifaceted. In the future, the API could also serve data to an integrated mobile application for tablet computers and smartphones that provides the exact same functionality and it would not be necessary to completely rebuild the core of application.

One of the key components of the API is the authentication system. I was very concerned about it early on in the application's development because I have found it to be one of the most difficult pieces to implement and it is critical to do so properly. An insecure authentication system puts user data at risk and seriously threatens the sanctity of the

application. After researching various methods of securing Node.js APIs, I determined that JSON Web Token (JWT) is a commonly used, supported, and effective solution to this problem.

JWT is an open internet standard (RFC 7519) that defines a methodology of securing and transmitting information between multiple parties [10]. It consists of a JSON object with a header, a payload, and a signature that is passed between the API and the API consumer (which is the application user in this case) and helps verify the identity of the API consumer [11]. The header of the object identifies that the object is a JWT and the specific encryption hashing algorithm that is being used. The payload contains statements about the entity like the name of the user, username, and user ID and may sometimes contain information about when the token was issued and when it expires. These token issue and expiration dates enables the API to require users to authenticate on a regular basis for added security. The signature portion of the token is made up of a hash of the previously mentioned header and payload in addition to a secret – which is only held by the API and is used to sign the JWT – using the encryption algorithm that is specified in the header. This signature is joined to the Base64URL encoded header and the Base64URL encoded payload by periods that separate the components. Here is an example of a JWT that is fully encoded, taken from the official JWT website [jwt.io](https://jwt.io) [10]:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjYoYZgeFONFh7HgQ

When an API consumer sends a JWT to the API, the API decodes the Base64 URL header and payload individually, identifies the encryption algorithm that was used to sign the token, and then uses its privately stored secret to decrypt the signature. If the signature decryption fails, the API will know that the token is no longer valid and reject the API consumer's request. If the signature decryption is successful, it will allow the consumer to make the request by using the information stored in the payload. This is the decoded version of the previous JWT [10]:

```
{
  "alg": "HS256",
```

```
    "typ": "JWT"
  }
  {
    "sub": "1234567890",
    "name": "John Doe",
    "admin": true
  }
```

It is important to note that important information like passwords should never be stored in the payload of a JWT because the payload is only encoded in Base64URL, which is easily decoded and is not secure. Instead, passwords should only be used to acquire the JWT from the API. When implemented correctly, the authentication user flow with JWT should resemble the following:

- 1) API consumer makes a POST request to the API's HTTPS protected authentication endpoint (like `https://api.yourapplicationhere.com/authenticate`) with their username and password included in the request headers
- 2) API receives the request and encrypts the user's password using a one-way encryption algorithm
- 3) API compares the encrypted password against the database's previously encrypted version (from when the user first signed up for an account) and determines if there is a match
- 4) If there is a match, the API creates the JWT with a few pieces of the user's information like username, name, and user ID included in the payload and returns the JWT. If there is not a match, the API denies the user's authentication request.
- 5) The user makes all future requests to the API by including the JWT in the header of the request and the API verifies the token each time and grants permission to access the data as needed

This flow exactly describes how the authentication system works in the API that I developed. I specifically used bcrypt to encrypt user passwords, as bcrypt is significantly slower than other commonly-used one-way encryption algorithms like MD5, SHA-256,

SHA-512, and SHA-3 [12]. According to Coda Hale, a software engineer at Stripe, modern servers can crack passwords that were encrypted with MD5 using brute force in a matter of seconds [12]. Since bcrypt is a much slower algorithm, Hale asserts that a password encrypted with bcrypt will take over decade to crack on modern servers, even if the password is lowercase, alphanumeric, and is only six characters long [12]. Additionally, the algorithm can be adjusted to suit the computational power of attackers in the future in keeping up with Moore's law because the computational expense of performing the algorithm can be adjusted [12].

## REST API Structure

As previously discussed, all REST APIs have URL endpoints that serve specific data or perform specific actions based on the endpoint and HTTP request method. API best practices maintain that these API endpoints should be resource-based and friendly to the API consumer [13]. If a resource can only exist within another resource like the comments for a blog post, the resource should be tucked into the endpoint of the parent resource; instead of `/comments`, use `/posts/:postid/comments` where `:postid` is the unique identifier of a particular blog post [13].

Additionally, REST APIs should take full advantage of the standard HTTP request methods of GET, POST, PUT, and DELETE [13]. GET requests should be used to retrieve information, POST requests for sending data with the intent of creating something new, PUT requests for sending data with the intent of updating something that already exists, and DELETE requests for deleting something. API endpoints may have similar names but should perform different functions based on the request method. As previously stated, the `/posts` endpoint should return all of the user's projects if receives a GET request but it should create a new project if it receives a POST request with the new blog post data.

Since the data for the project management application is based on users, projects, tasks, and subtasks, it was logical to make these the resources for the API. However, each of these resources does not deserve its own separate endpoint. Tasks, for instance, can only be consumed in reference to a project; there is not a single instance in which a task is not associated with a project. This is also the case for subtasks in relation to tasks. Based on

the API design best practices, it made sense to split these resources up into the following endpoints:

- `/users`
- `/projects`
- `/projects/:projectid`
- `/projects/:projectid/tasks`
- `/projects/:projectid/tasks/:taskid`
- `/projects/:projectid/tasks/:taskid/subtasks`
- `/projects/:projectid/tasks/:taskid/subtasks/:subtaskid`

The `/authenticate` endpoint was added in to this list of endpoints as well because the authentication functionality does not fit into `/users`. The `/users` endpoint is used to register new users, retrieve user data, update user data, and delete user accounts. All of this actions uses up the available HTTP request methods so a new endpoint needed to be created for the specific purpose of logging users in. I set up the authentication endpoint and made the log-in function able to be requested via a HTTP POST with the user email address and password included as the headers of the request. I determined that this should be performed via a HTTP POST request instead of a HTTP GET request as users would be sending information with the intent of creating something new: a JWT that they can use to access the rest of the application.

## DevOps

When the API started to take shape, I decided that I needed to deploy the database and the API to their own individual servers. Platform as a service (PaaS) server hosting providers like Heroku make hosting incredibly easy by taking control of the server setup and maintenance process for you. This is not what I had in mind. I wanted to learn how to deploy web applications from the ground up and gain Development Operations (DevOps) experience from doing so. Virtual Private Servers (VPS) providers allow users to completely control and manage their own remote virtual servers that are located in server farms around the world. I decided to use Linode for my VPS solution because it offered the best server specifications for the money at the time in comparison to its competitors.

Provisioning the database and API servers on Linode was a painless process. All that was required of me was clicking through prompts to select the server operating system, physical server location, and the naming scheme. I chose Ubuntu for the server operating system since I had the most experience with its particular flavor of Linux and I chose the data center in Dallas, Texas because it offered the fastest download time for my physical location even though the Fremont, California data center is much closer. After both of the servers were provisioned, I had to set up the SSH keys on my laptop and SSH in to each of the servers to begin securing them and installing the necessary software.

I followed Linode's guide to securing servers which included setting up a limited user account, installing Fail2Ban to ban IP addresses that attempt to log in to the server unsuccessfully too many times, and configuring the firewall [14]. The basic iptables ruleset that was provided in the Linode guide was extremely helpful in getting past some of the confusion that I had with the iptables syntax and helped ensure that the necessary server ports were blocked.

Installing the necessary software on each of the servers varied in difficulty. MongoDB was very easy to set up because the installation process amounted to a few shell commands; the API itself manages the database schema with Mongoose so it was unnecessary to do much more than that. Setting up the Node.js API on the server was significantly more difficult as the setup process was lengthier and more intensive. This is partially because the API is public-facing (server port 80) and needs to be accessible without risking the security of the server.

Running the API on port 80 was a significant obstacle to making it publicly accessible. All ports fewer than 1024 are protected ports and can only be accessed by root users, meaning that the API's build and runtime application, Node.js package manager (NPM), has to be run by a root user [15]. The API can be run from a higher port without root access, but API consumers would have to specify the exact port whenever they make a request, e.g. `https://api.yourapplicationhere:1234/posts`. This is not an ideal user experience, so I ruled this option out. Running applications in root on a production server is not a suitable alternative either because it is not a good security practice; if there is a bug in the API or in Node.js in general, an attacker could take

advantage of the vulnerability and gain full access to the server [16]. A better alternative is to allow Node.js to bind to all of the protected ports on an application level through the use of the `setcap` command [15]. This approach completely resolved the issue but it needs to be run again any time that Node.js is updated. I had the tendency to forget about this requirement and completely broke the API on the server on several occasions because it was not able to access port 80.

Running the API on port 80 is not the only measure that needed to be taken to ensure that the API is publicly accessible. If the server was shut down either intentionally or unintentionally, the API would not come back up automatically when the server was started again. I used PM2, an open source process manager built for Node.js, to keep the API alive forever. PM2 also provides a lot of other useful functionality like load balancing and application reloading without downtime in case a change needs to be implemented without negatively impacting the users [17].

Giving the web application a user friendly name other than the server IP address was a priority as well. I bought a domain name off of GoDaddy and configured the domain registrar's settings for the domain to use Linode's name servers. When I went to Linode's DNS Manager to set it up, I received several error messages indicating that my new domain name already existed in Linode's database. After contacting support and verifying that I was in fact the owner by changing the WHOIS information, I was able to finish setting up the domain name for the API and the future frontend server.

## Automated Testing

Testing is incredibly important to software development. If a piece of code hasn't been tested, it should be considered non-working. I used the Mocha JavaScript testing framework and Chai assertion library to perform unit tests in order to remain true to this development ideology. Chai is used to make assertions about the outcome of functions that forms the tests that Mocha runs. A test on the data returned from a GET request on the `/projects` endpoint would include Chai assertions that the HTTP status code is 200 (everything is okay), the data returned is an array of JSON objects, and that each of the JSON objects that were returned have data elements like an ID, project name, project owner, etc.



These tests are integrated into NPM and can easily be run with a single terminal command. However, it is still a little unwieldy to run all of the tests as regular part of development, so I wanted to automate the process. I already used a private GitHub repository to manage the code and decided to take advantage of Travis CI's integration with the platform. When combined with GitHub, Travis CI automatically runs the tests on the codebase whenever changes are pushed up to the repository and then sends out the results of the test to the development team. The optional Travis CI status tag on the GitHub repository will also change to indicate the test results. Travis CI has the capability of deploying the confirmed working code to the production server if all of the tests are passed as well, establishing a continuous delivery pipeline. Since I did not use one of the supported PaaS options for hosting the API, there was a lot more work involved to get Travis CI to deploy to my production server. Rather than spending the time to get the system set up, I instead opted to run manual deployments.

Travis CI works very well if you have well-written tests, but is essentially useless otherwise. I did not put enough time into researching how to properly use Mocha so I only wrote tests for GET requests on the endpoints, completely ignoring POST, PUT, and DELETE requests. This left a giant gap in my tests that let bugs through on occasion. I still manually tested any changes that I made before committing the code and pushing it to the GitHub master repository but I sometimes missed a few test cases. The logical fallacy behind allowing these testing gaps to remain was that testing did not amount to “actual work” on the program; I could either spend time getting the testing framework to fully work or I could complete several new features. This was misguided thinking as I spent a significant amount of time fixing minor bugs that would have been caught by a properly implemented testing framework instead of working on those new features. The partially-realized testing framework became even more of a hindrance due to the state of the authentication system. I did not program a way for the testing framework to authenticate itself before querying protected API endpoints so I had to manually retrieve a JWT for it and add it into the API's configuration file before committing the code and pushing it up to the master repository. This completely defeated the purpose of using an automated testing framework and I wasted more time by not fully implementing the tests.

# Developing the Frontend

## Frontend Background & Bracing for Change

With the REST API completed, I was able to begin developing the frontend of the web application. The frontend is the layer of the application that users interact with. Any data manipulation that the users perform here is communicated to the API and then reflected in the database. The user interface has to be incredibly compelling and friendly to the user as the user's judgement of the overall application is made based on their experience with this particular layer; the API can be well designed but that does not matter if the user interface does not delight. Per the MEAN stack, the frontend of the web application was built on AngularJS with Bootstrap. AngularJS provided a framework to retrieve data from the API and added the needed interactivity to the web application while Bootstrap ensured that the web pages were responsive for a variety of screen sizes.

Both AngularJS and Bootstrap released new beta versions, AngularJS 2 and Bootstrap 4, over a month into the development of the frontend of the web application.

Unfortunately, these newer versions were incompatible with the now old versions that I was using, so all of the code that I had already written would need to be migrated over to the newer version [18], [19]. Migrating to each of the betas would involve a complete rewrite of the frontend, which would set me back significantly. In my experience, beta versions typically have more bugs than the final version and I was not sure that I wanted to perform a lot of extra troubleshooting. I experimented with the Bootstrap 4 beta when it first was announced anyways so that I could learn about some of the changes firsthand. Several of the examples on the website were completely broken which discouraged me from exploring further. There was no point in performing a rewrite of the HTML views so that previously working code could be replaced by a newer version that did not work at all.

I ruled out the AngularJS 2 beta as well as it would completely change the structure of the application due to the framework's new focus on components and the change in

programming language [20]. AngularJS 2 focuses primarily on the use of JavaScript alternatives TypeScript and Dart and I was not familiar with either language. The traditional JavaScript was still an option, but the majority of the tutorials for the newer version on the official site were written in TypeScript. There were simply too many different changes between the versions and it even impacted several of the AngularJS libraries. The frontend application takes advantage of several libraries, primarily Chartist.JS and Angular UI Sortable, in order to provide the necessary functionality for the application. These libraries did not support the newest AngularJS version yet, so switching would mean that I would have to abandon these libraries which was an unacceptable cost.

## UI/UX Design

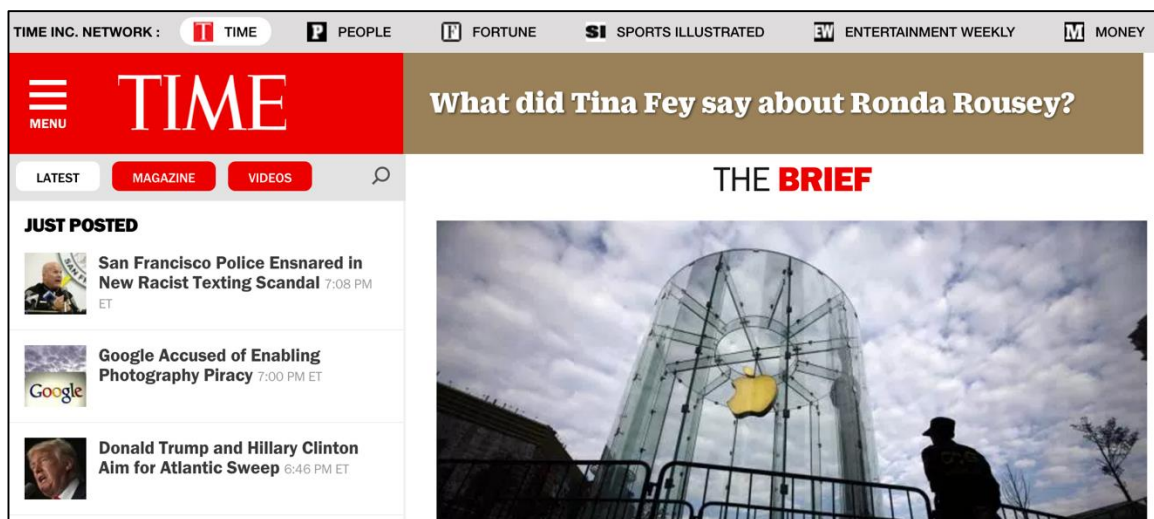
There are many tutorials and guides available that provide instruction to developers on building AngularJS web applications but these guides do not impart the knowledge of creating compelling and accessible user experiences. This is something that I struggled with for several months while developing the frontend of the web application. The frontend is the primary touch point that users interact with and a good experience here drives users back to the site.

One of the most important considerations of frontend development is the frontend's ability to cater the various screen sizes of internet-capable devices. In May 2015, Google reported that its users perform searches on the company's search engine on mobile devices more than desktop computers in over ten countries, including the United States and Japan [21]. The future of the web is responsive and as such, I needed to use a framework like Bootstrap to cater the many different types of internet-capable devices.

One of the more difficult aspects of "mobile-first" development is creating a navigational system. Traditionally, a site navigation bar is included at the top of the webpage and includes links to some of the more important and heavily used areas of the site. I originally planned to use this system but I also considered a vertical navigation bar that is placed on the left side of the page as well. I experimented with this layout and found that it was cumbersome to interact with on smartphone-sized screens. Smartphones in general are an imperfect device to use the application because there is not enough screen space to meaningfully interact with the Kanban boards with the exception of viewing

tasks. This is not something that can be resolved through advancements in the design of the user interface but the vertical navigation bar made it even more difficult to use so the horizontal bar was brought back.

Horizontal navigation bars have their own set of issues. They work perfectly on larger screen sizes but the links on the bar do not scale down well. The common design response to this issue is what is known as the hamburger button. The hamburger button, also known as the hamburger menu, is characterized by three stacked horizontal lines that vaguely resemble a hamburger, the top and bottom bars representing the hamburger bun and the middle bar representing the hamburger patty (See Figure 4.1). The button is typically used to hide parts of a navigational menu away from the viewer but still provides the functionality in the case that the user needs it. Luis Abreu, a Senior UX designer at Luis Abreu Ltd., asserts that hamburger menus should be avoided because they introduce navigational friction by forcing users to open a menu in order to reach their destination and the items that are hidden inside are ignored as a result [22].



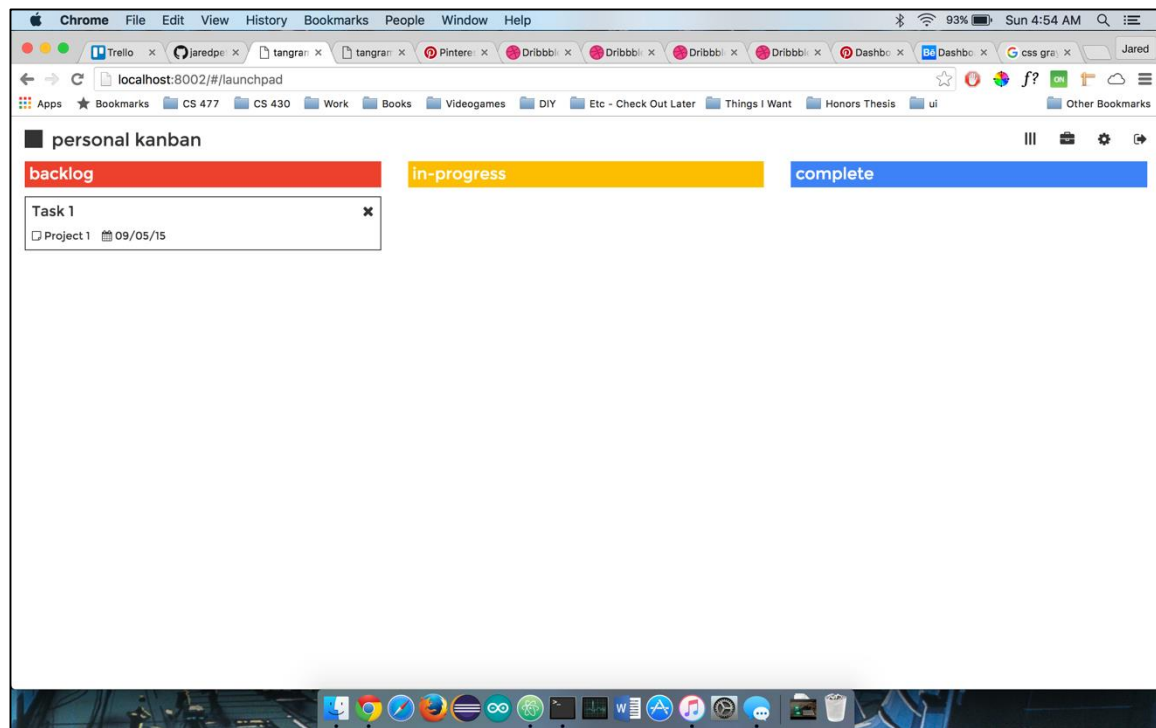
**Figure 4.1** Hamburger menu on the front page of the Time Magazine website

Rather than use a hamburger menu, I experimented with several different techniques to make the navigation bar usable on smaller screen sizes. One of the approaches that I explored was changing the link icon sizes and limiting the number of icons on the menu bar. This was not ideal, as it limited the usability of the site further and was not aesthetically pleasing; the smaller icons appeared to be out of place next to the large page title. I also tried to change the wrapping behavior of the icons on smaller screen sizes. In

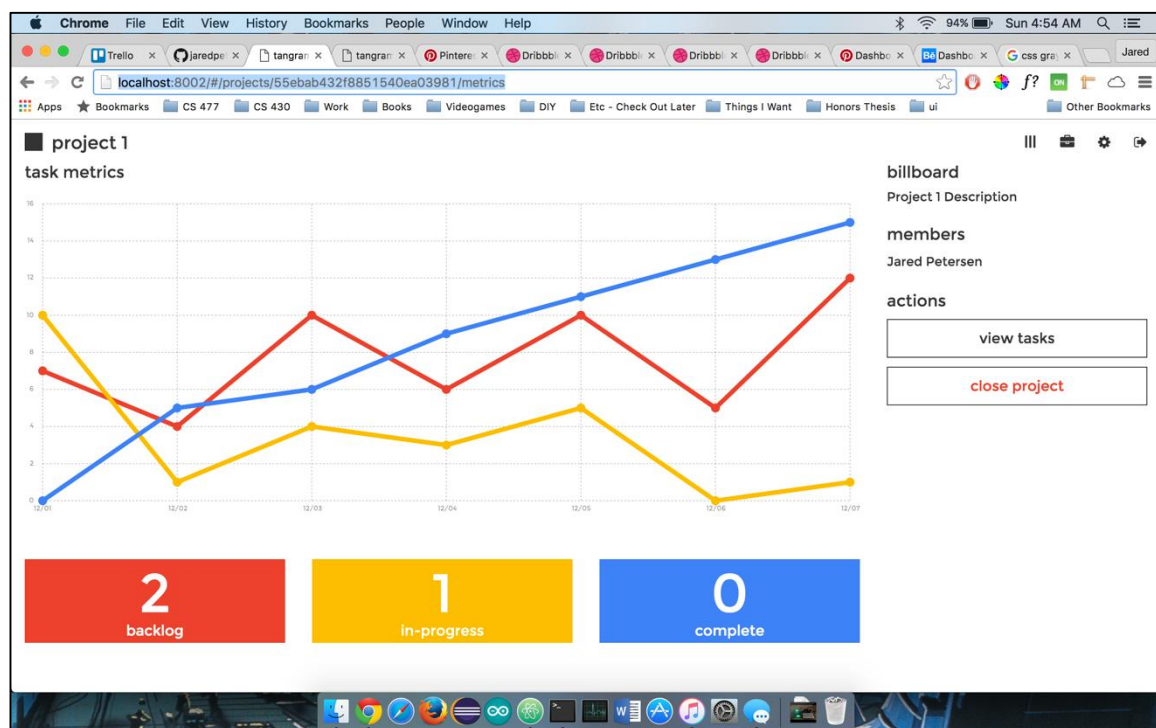
Bootstrap, modifying the navigation bar template to not include a collapsing hamburger menu completely breaks the responsive navigation link-wrapping functionality.

Modifying this further so that the navigation links are slide directly underneath the page title when they no longer fit on one line of the navigation bar produces a thicker bar that is still visually appealing and compensates for longer page titles.

Another important aspect of frontend web development is the overall design and visual identity of the website itself. Since the web application is focused on organizing projects, I wanted it have an extremely minimal and modern design that imparted a feeling of cleanliness. One of the key pieces to this centered around flat design, an interface design style that does not use any design elements that give the illusion of three dimensions [23]. Flat design traces its roots back to the International Typographic style of the 1940s and 1950s but was more recently popularized by Apple's mobile phone and tablet operating system [23]. In keeping with this style, tasks would take the form of white rectangles that were visually similar to index cards. Kanban board columns would be colored with primary colors to indicate the status of the card in a bright and visually appealing manner. The design would very closely match the original wireframes that I created because the wireframes essentially implemented flat design from the very beginning. Figures 4.2 and 4.3 show the first version of this design plan.



**Figure 4.2** First iteration of the project Kanban board



**Figure 4.3** First iteration of the individual project dashboard

The red, yellow, and blue coloring was originally chosen to accommodate colorblind users. I took advantage of the Spectrum Google Chrome extension to view the application from the perspective of the various different types of colorblindness and sought advice from a colorblind colleague. Unfortunately, informal feedback from my peers indicated that the color choice was not ideal. The colors were contrasting enough for those who are colorblind but the design did not clearly communicate that the colors represented the task statuses. This led me to switch the coloring scheme to red, yellow, and green so that the task statuses would be strongly associated with stoplights. I was originally wary about using red and green together as people that are red-green colorblind comprise the largest segment of colorblind people [24]. This color combination is generally not recommended as a result, but they can be used if the colors are bright, have high contrast, or have differences in hue and saturation [25], [26]. Spectrum was extremely useful in validating these colors to ensure that the site is accessible to colorblind users.

Only using red, yellow, and green for task statuses, black for text, and white for the general background of the page seemed visually appealing at first, but limited the layout of the application. The web application elements blended together to the point that any information that was not surrounded by bold colors was visually left behind. Delineating these elements was very difficult without the use of unattractive black borders, particularly on the individual project details page. Another approach needed to be taken.

I redesigned the frontend by significantly reducing the amount of whitespace on the page and taking advantage of neutral colors – particularly shades of gray – to delineate the sections of the application. Rather than using black element borders, the contrast between the overall background color and the color of the elements formed a sort of border that gave each element its own visual identity. This made it easier to add new elements and determine how the elements should be arranged visually on the page during development. Please see the appendix for the most recent evolution of the design.

## **Displaying Data via Charting Libraries**

Building the frontend of the web application was challenging from a design perspective and a technical perspective. One of the most technically difficult features to develop was charting the task information with the Chartist.JS charting library. The main chart that I

wanted to include in the individual project details page was a simple line chart that shows the number of tasks in each status group over time. I thought that this would be a good start and that I could build off of it later with the addition of a burndown chart.

When I first began to develop the API, I did not think about storing the historical task statuses; tasks only had a single status indicator that indicated the task's current status. I had to modify the task Mongoose schema and API functions so that the task indicator would be an array of JSON objects that store a date ID and the status for the date. Any time the task status changes, the modified API would add a new record for the day or modify the existing record if the task status had previously been modified that day. Every task would have one status record each day that the task status had been modified.

To make it easier for API consumers to retrieve the data, I needed to create a new endpoint that performs a query that counts the tasks in each of the columns for each of the days. I pored through the MongoDB documentation and the StackOverflow posts and it became abundantly clear that performing such a query in MongoDB would be more complicated than its SQL alternative. Queries that are more complex than the equivalent of a simple SQL SELECT statement are not ideal in MongoDB as the database's more advanced querying syntax is confusingly difficult to master.

The query that I wrote was exceptionally long but it did return back the requested results. Unfortunately, those results were useless. Take the following example: three new tasks were created on April 19. The next day, one of the three tasks was moved to another column. From the perspective of looking at each task individually, there are clear timestamps that indicate where the project is at a given time; two of the tasks only have a Backlog status and the third task has a Backlog status on April 19 and an In-Progress status on April 20. The database query is counting up all of the tasks from a status-first perspective. It counts all of the tasks for each status and day accurately and returns the following result:

```
{
  "historical_status":
  {
    "backlog":
```



```

    [
      {
        "_id": "2016-04-19T00:00:00.000Z",
        "total": 3
      }
    ],
    "in-progress":
    [
      {
        "_id": "2016-04-20T00:00:00.000Z",
        "total": 1
      }
    ],
    "complete": []
  }
}

```

This result is strictly correct, but it does not take the actual progression of tasks from one status to another into account. The user cannot assume that the three tasks that were in the Backlog on April 19 are still there on April 20 or that there were not any tasks in the Backlog on April 20 at all. There is no way for MongoDB or any relational database to determine how many tasks are in any of the columns because the two tasks that are actually still in the Backlog do not have a recent timestamp. The query would work perfectly if each task received a new status timestamp every day but there is not a good methodology to accomplish this.

The obvious solution to this problem is a nightly process that adds a new record to the status array if its respective task did not already have one for the day. This is inefficient and is not scalable, so it is not a solution worth pursuing. The only solution that I have come up with for this new issue involves retrieving all of the tasks for a given project and then iterating through them on the basis of date. A seven-day period is a good first prototype, so in that case the outer loop would need to cover the previous seven days and another inner loop would need to iterate over the tasks and count the number of tasks in each of the columns for the outer loop's day, essentially constructing a daily task

calendar from the gaps in history. I have not yet implemented this strategy yet, so I have not been able to determine its validity. Once the counts have been reached, the endpoint would need to output the data in a friendlier format than the one returned by the MongoDB query. Chartist.JS in particular requires the charting data to be in the form of several arrays: one array for the X-axis chart labels and another array of arrays that store the series data for the Y-axis of the chart. This format is a logical way of storing chart data, so the output data would probably take this form instead.

# Software Engineering & After Action Review

## Software Engineering Methodology

The application has come very far since Summer 2015 but there is still a lot of work left to do, which is true of any software project. As time goes on, new features are included and bugs are discovered. Software engineering is an iterative process and there is a sort of beauty in continuously improving software and one's own skills. While the previously set-upon qualities of the minimum viable product have not technically been met because there was not enough time to finish implementing the charting functionality, the application still serves as an alpha version that provides an indication of where it will one day stand.

The software development methodology that I used to develop the application, Agile Kanban, was very effective in organizing the process. I have used the more generally well-known Agile Scrum development methodology in the past but it was not a great fit for the characteristics of this project. Agile Scrum relies on the use of Product Owner and Scrum Master positions to communicate the vision of the product to the team and facilitate the development process [27], [28]. Since I was a single-person development team, I would be unable to take full advantage of these roles. Agile Scrum also uses fixed length sprints, typically one to four weeks long, to break development down into manageable pieces and focus the work [29]. I was not able to work on the application full-time so my development schedule did not line up with these traditional sprint cycles. Agile Scrum's formal procedures and clearly defined and structured workflow works for large teams of full-time software engineers but it would only hinder my ability to develop the application.

Agile Kanban is an alternative to Agile Scrum that removes these formal positions and turns development into a continuous flow [30]. Rather than using fixed length sprints,

Agile Kanban uses a Kanban board to track progress. The team can add new user stories, assign user stories to themselves in line with team's stated objective (sometimes with the help on an Agile coach), and work the user stories through the columns on the board [30]. The Kanban board columns are customized to fit the development team's software deployment process and includes a limit on the number of stories that can be in each column, called the Work in Progress (WIP) limit. These limits highlight bottlenecks in the development process, giving the team the opportunity to rectify these bottlenecks in order to guarantee that quality software is being delivered to the end user as fast as possible [30]. When a story is completed and is run through the necessary testing measures, it can be deployed immediately instead of waiting for the end of the sprint like in Agile Scrum.

The Kanban board for this particular project took many different physical forms but consistently used a Backlog, In-Progress, and Complete status columns. In the beginning, I used 8" x 6" oversized sticky notes for the user stories and Kanban column labels and stuck them to a large open wall in my apartment. I really appreciated the tactile feel of the sticky note cards and the large board motivated me to work on the application further. However, this physical board was not very useful when I was trying to work on the application outside of the apartment. I used Trello to digitize the board so that I could bring it with me anywhere. It is still not effective for managing projects but it executes electronic Kanban very well. As soon as the basics of the frontend of the web application were complete, I started to use the application that I was building to help manage its development (dogfooding).

## **Mistakes and Future Development**

If I were to continue developing the web application, I would like to make several modifications to the API and UI layers. Most of these modifications are relatively minor in terms of scope, as there will always be a million little things that will need to be taken care of. The major changes generally relate to the almost-finished status of the minimum viable product, the current user experience, and fulfilling the overall vision of the application.

The minimum viable product is essentially complete but it is missing a few components. As previously mentioned, the task chart on the Individual Project Details page is not

currently working and needs to be fixed. Additionally, the application does not provide a method of adding new users to a project. If a new user was somehow added, the application would support that user but it does not provide a specific function to do so. Users are also unable to change their password or email address as well, which is something that will need to be implemented if the application is made public. These issues are minor and should not be difficult to finish implementing; there just was not enough time to do so.

The development experience can be improved as well, particularly in the area of testing. A limited form of automated testing as already been set up but there is not full test coverage. The rest of the tests should be created in the future as a priority in order to reduce the amount of bugs that are introduced into the codebase. It would be ideal if the existing automated testing system could be expanded in the future to also be integrated with the production API server. As it currently stands, when a commit is pushed to the master GitHub repository Travis CI runs the automated tests. Travis CI also allows developers to deploy their working code after all the tests have passed so that the pipeline from developer to user is completely continuous and never stops flowing.

Moving forward, there are a few additional enhancements that I would like to make to the application. One of these enhancements is tighter API security. The password hashing system and JWT authentication system are a good start but more can be done in this area. In the current JWT implementation, the tokens do not become invalid until they expire. If a token became compromised, the attacker would have full access to the application and there would not be a solution to stop them. Some JWT libraries support the addition of a unique token identifier called a JTI to the token's payload, which allows system administrator to lock out tokens with the specific identifier [31]. The library that I used, node-jsonwebtoken, did not support the JTI natively until very recently so I was not able to include it when I first started building the API [32]. Now that JTI is a standard feature of the library, it would be beneficial to integrate it into the API. Another minor change is protecting all of the production servers via HTTPS. I looked into the process of doing so before setting up the servers on Linode but decided to put it off until the application was close to going live. This was not part of the original minimum viable product specification, but its importance to user data security warrants its inclusion.

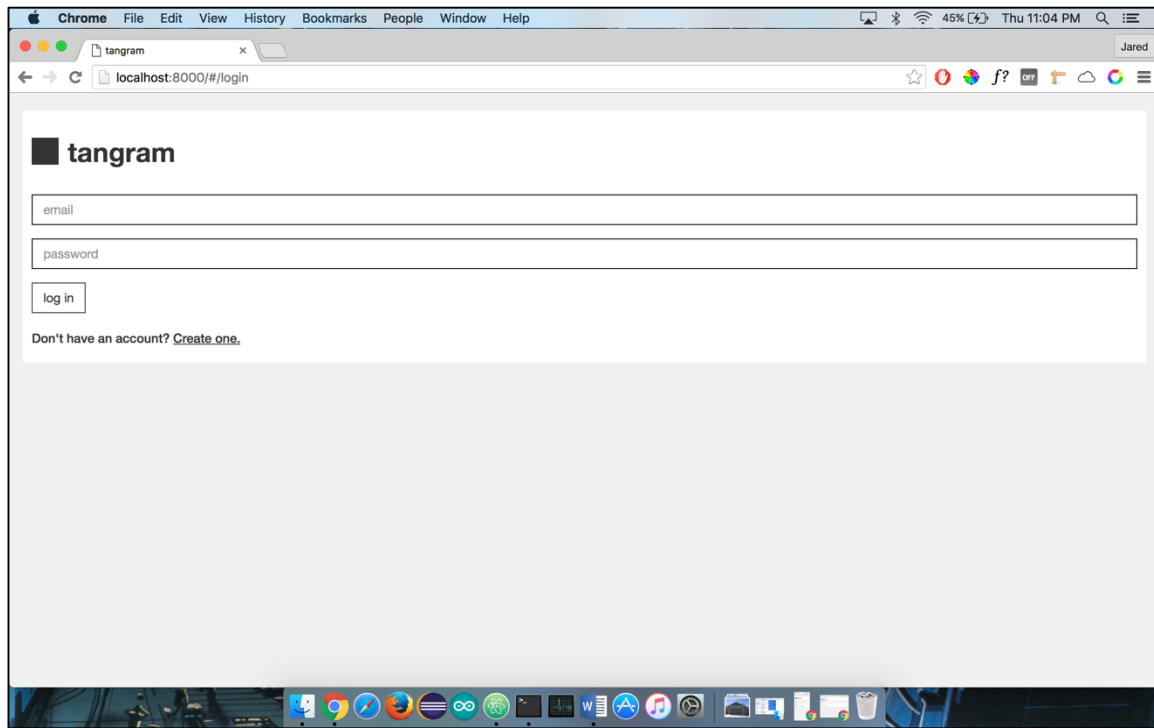
It would be best for the application if features like a calendar and chat system were included after the minimum viable product was released. Other features like real-time data from the API would be ideal as well so that users do not have to refresh the application manually to confirm that the data has not changed. Flashy animations for functions like editing task details would really add another level of polish to the application that would bring the application up to more modern expectations of web user interfaces. Since AngularJS and Bootstrap are coming out with completely new and not backwards-compatible versions, it might be worth spending the time to migrate the application over as well once both platforms gain more support from the development community.

## Conclusion

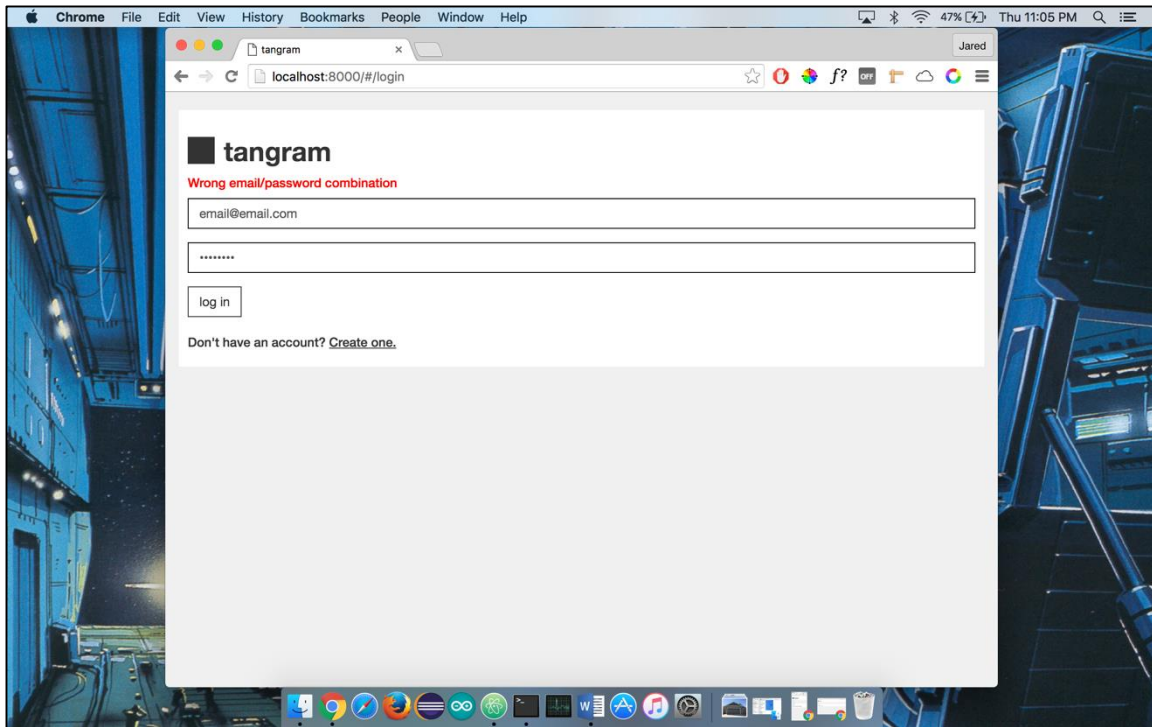
I learned more about software engineering over the past year than I have from the previous three years of academic instruction. The hands-on, real-world experience cemented and exemplified the ideas of why Agile software development is so much better than Waterfall, why testing is so important, and why the architecture of applications matter. I was able to build a full-stack web application through copious amounts of research and experimentation and gain some DevOps skills in the process through hosting it on a VPS. The skills that I have developed and the experience that I have gained has made me a better software engineer and I will be able to use this knowledge moving forward in my career.

## Appendix

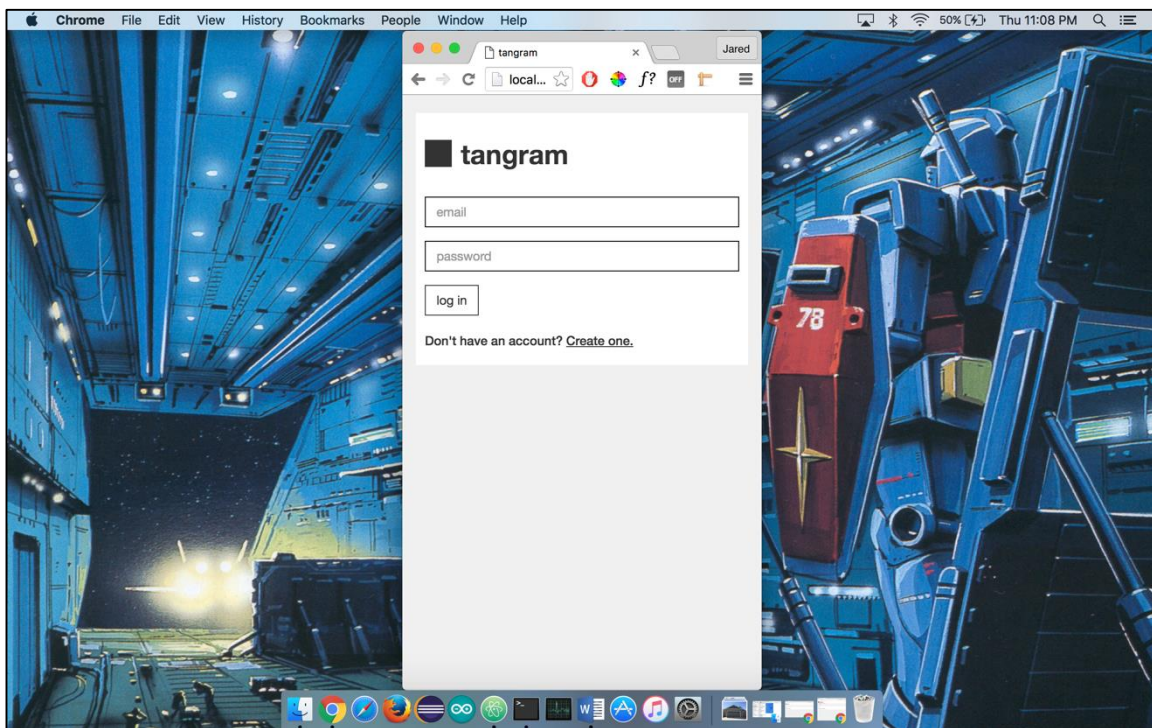
# Application Images



**Figure A1.1** Login (desktop)

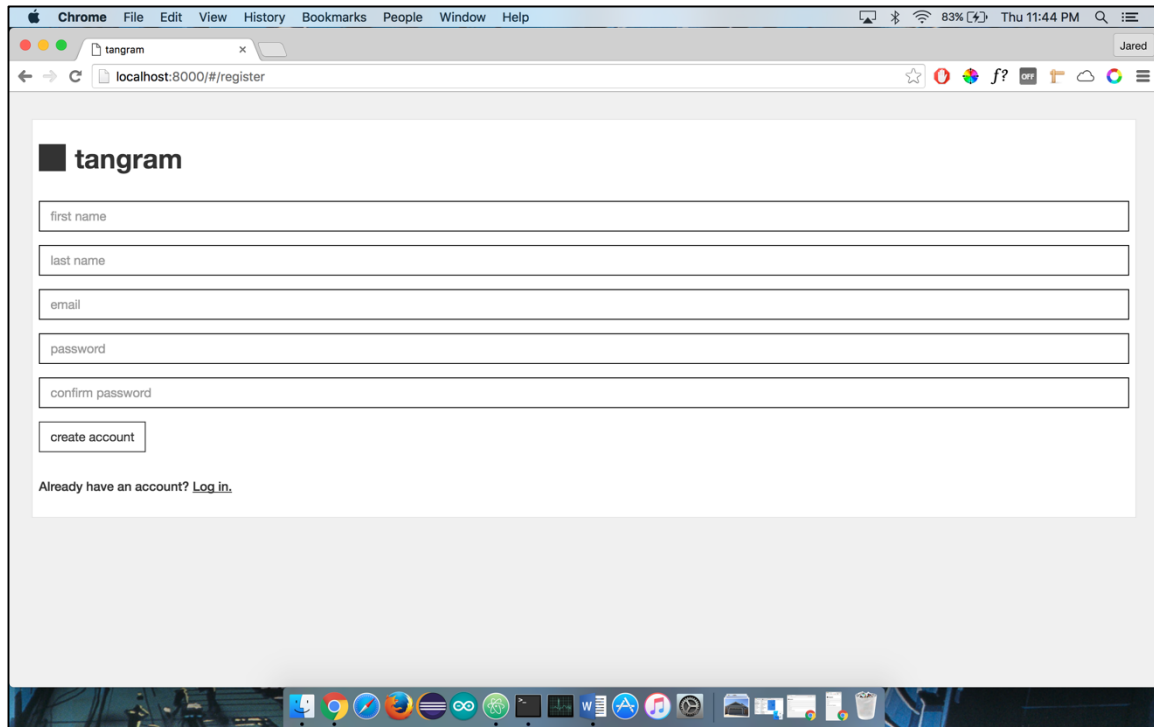


**Figure A1.2** Login (tablet)

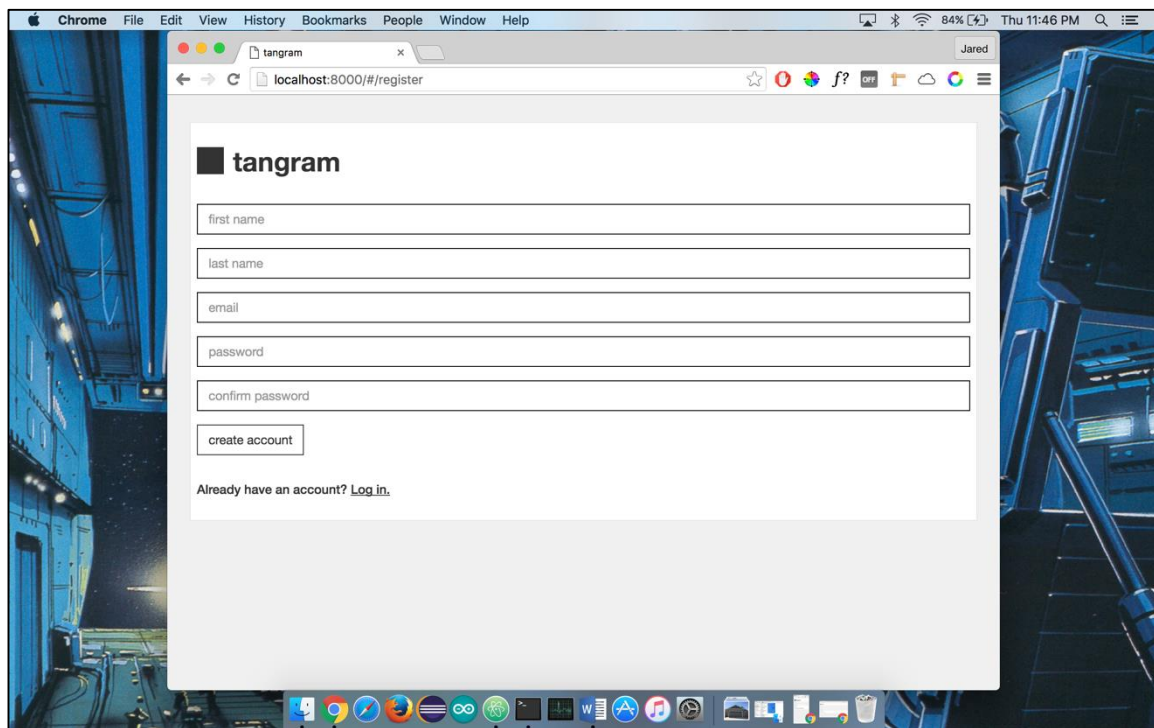


**Figure A1.3** Login (smartphone)

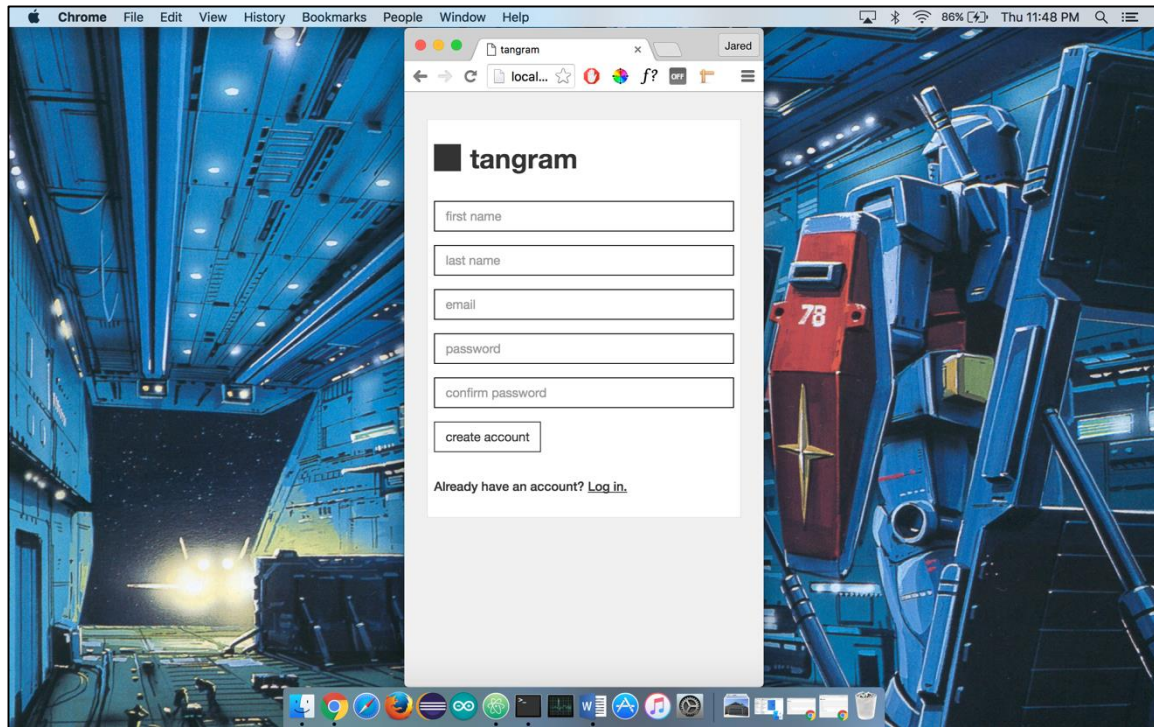




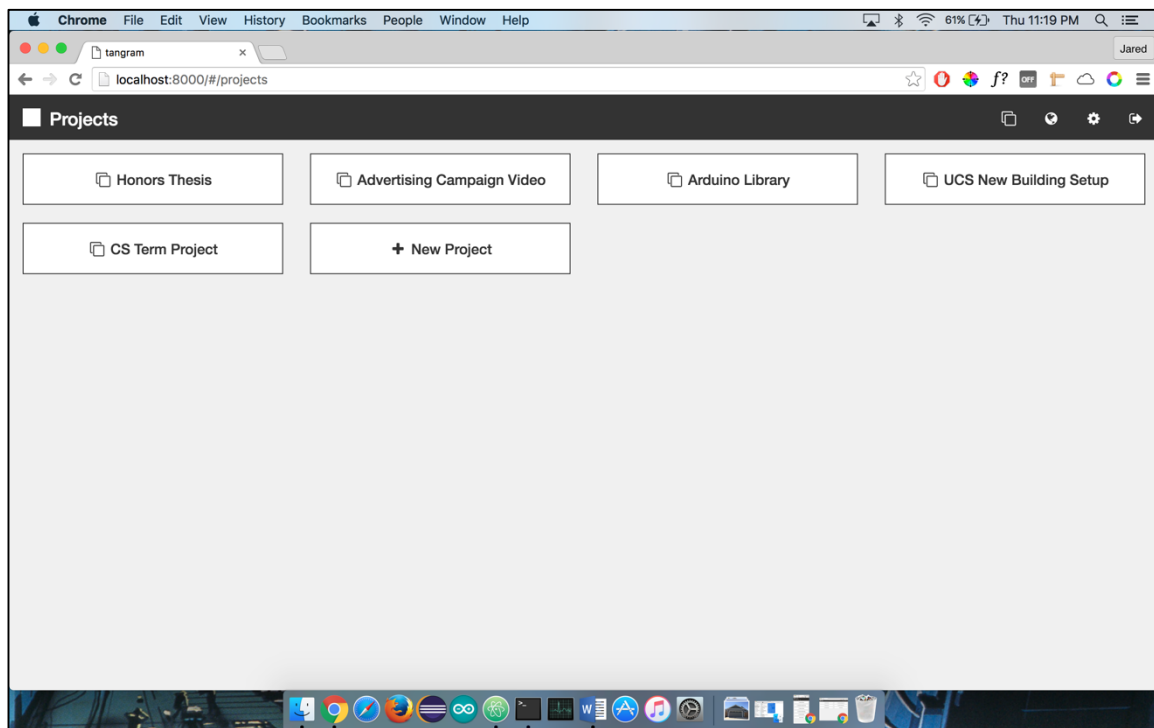
**Figure A1.4** Registration (desktop)



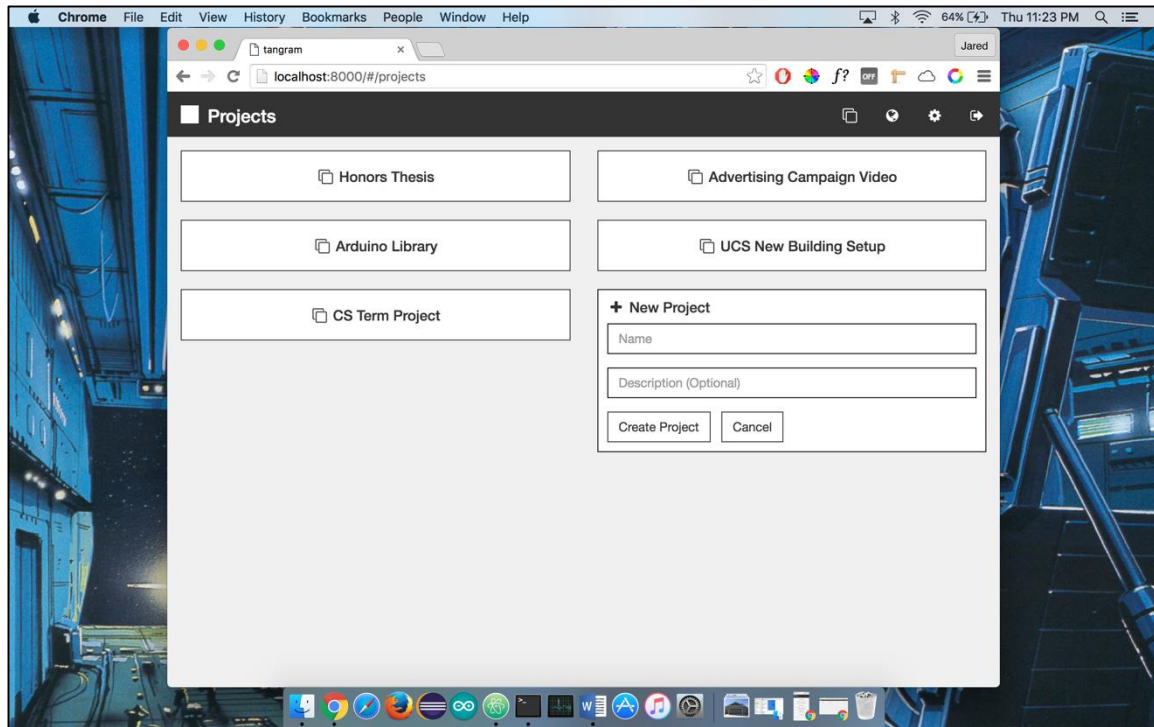
**Figure A1.5** Registration (tablet)



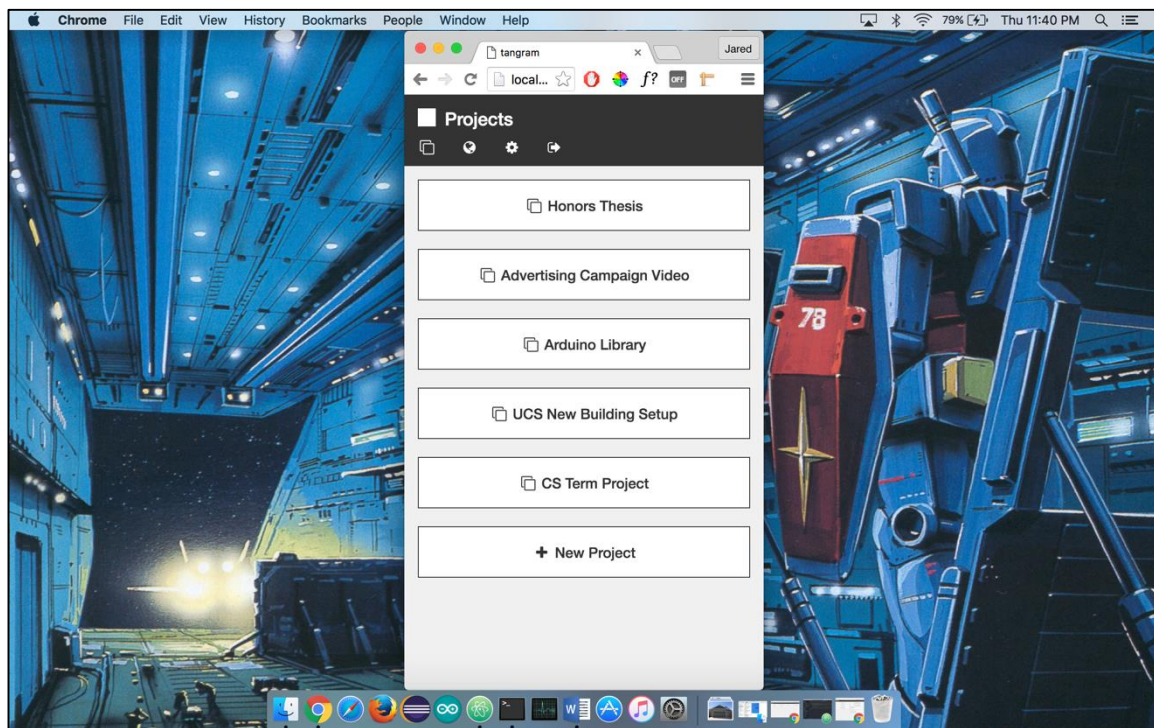
**Figure A1.6** Registration (smartphone)



**Figure A1.7** Multiple projects (desktop)

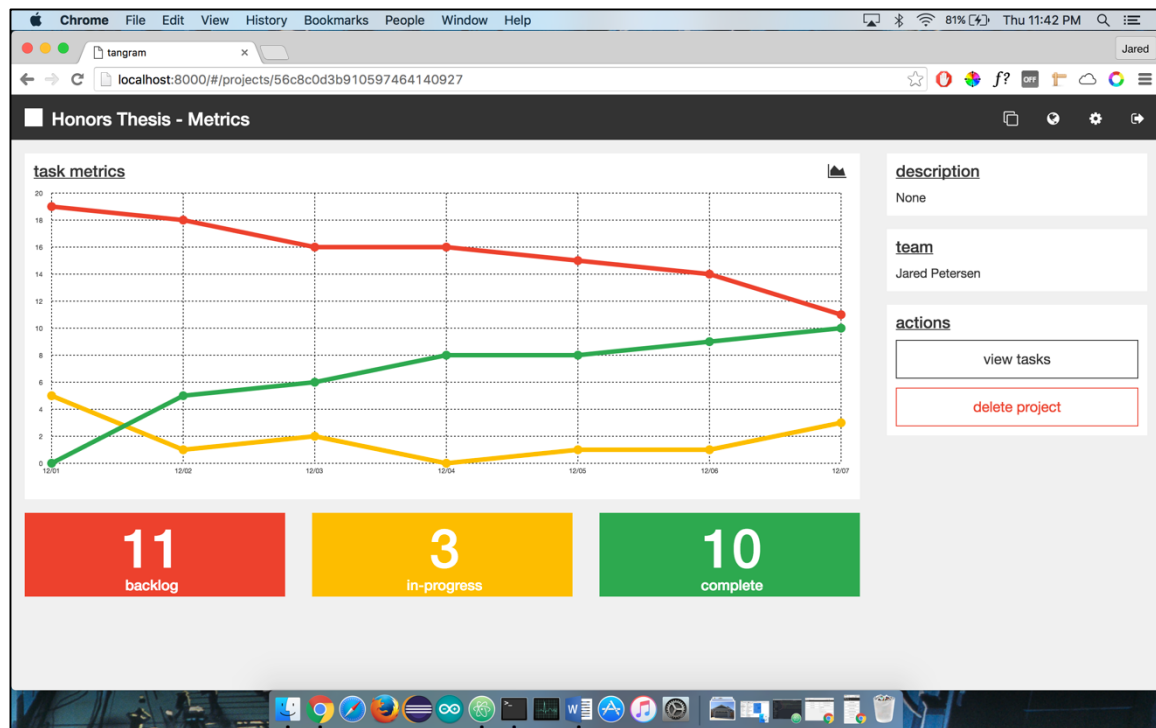


**Figure A1.8** Multiple projects (tablet)

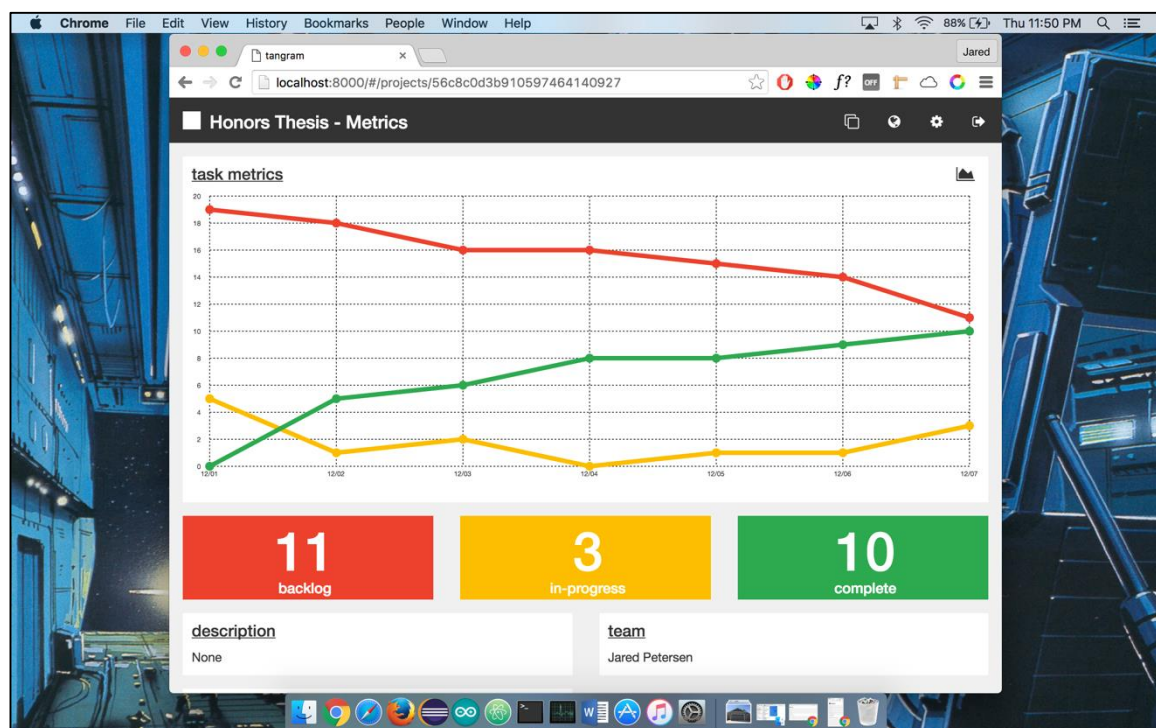


**Figure A1.9** Multiple projects (smartphone)

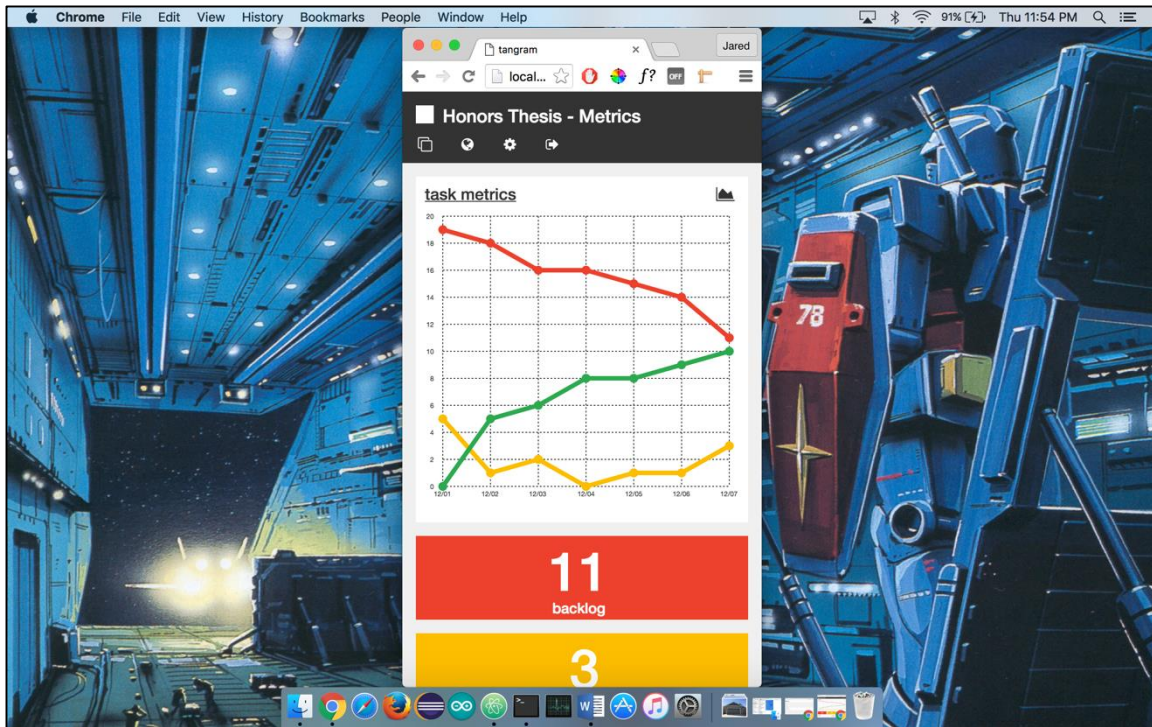




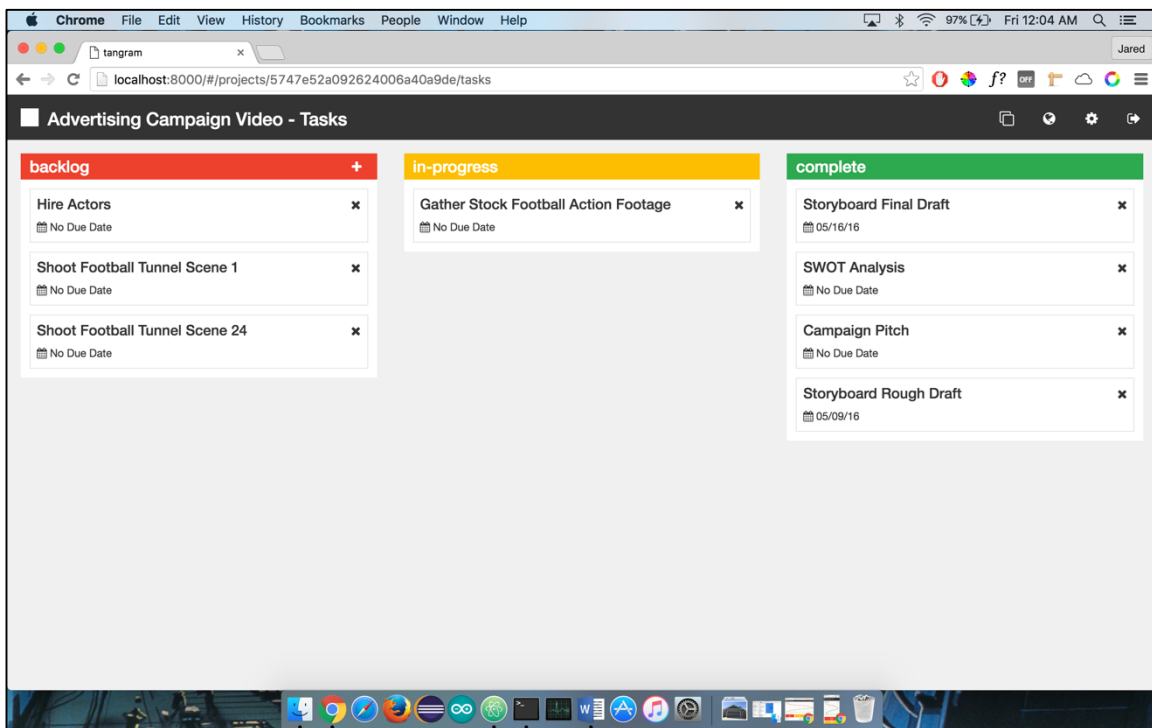
**Figure A1.10** Individual project with inaccurate charts (desktop)



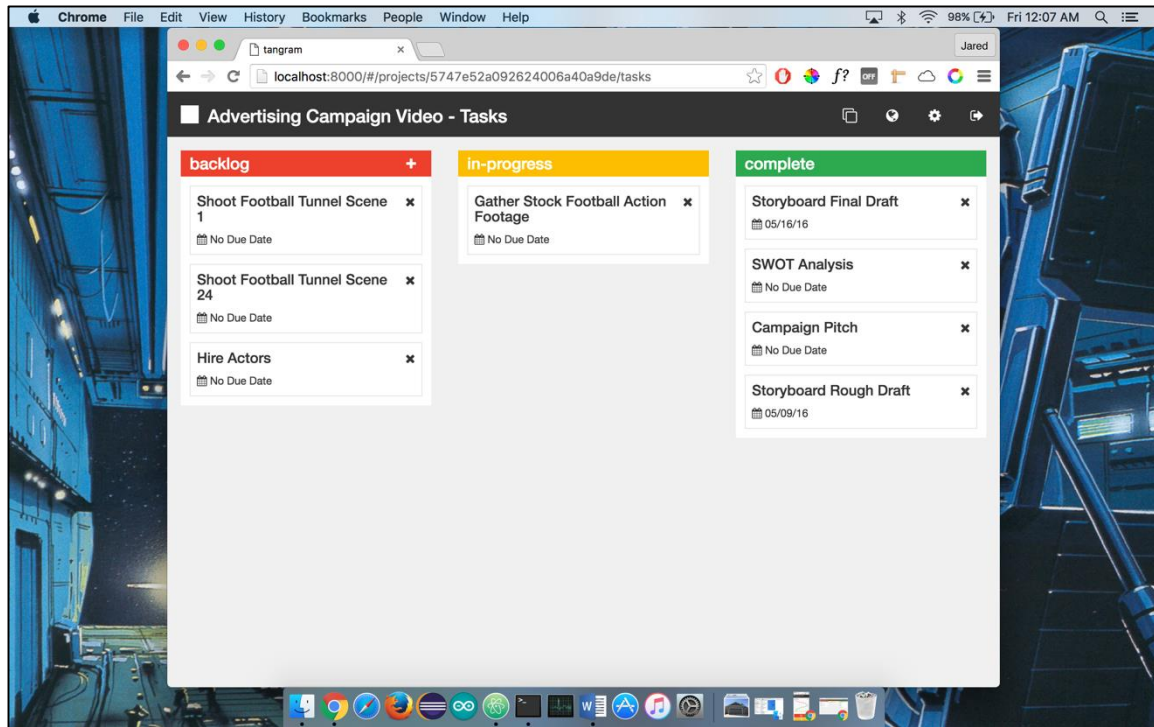
**Figure A1.11** Individual project with inaccurate charts (tablet)



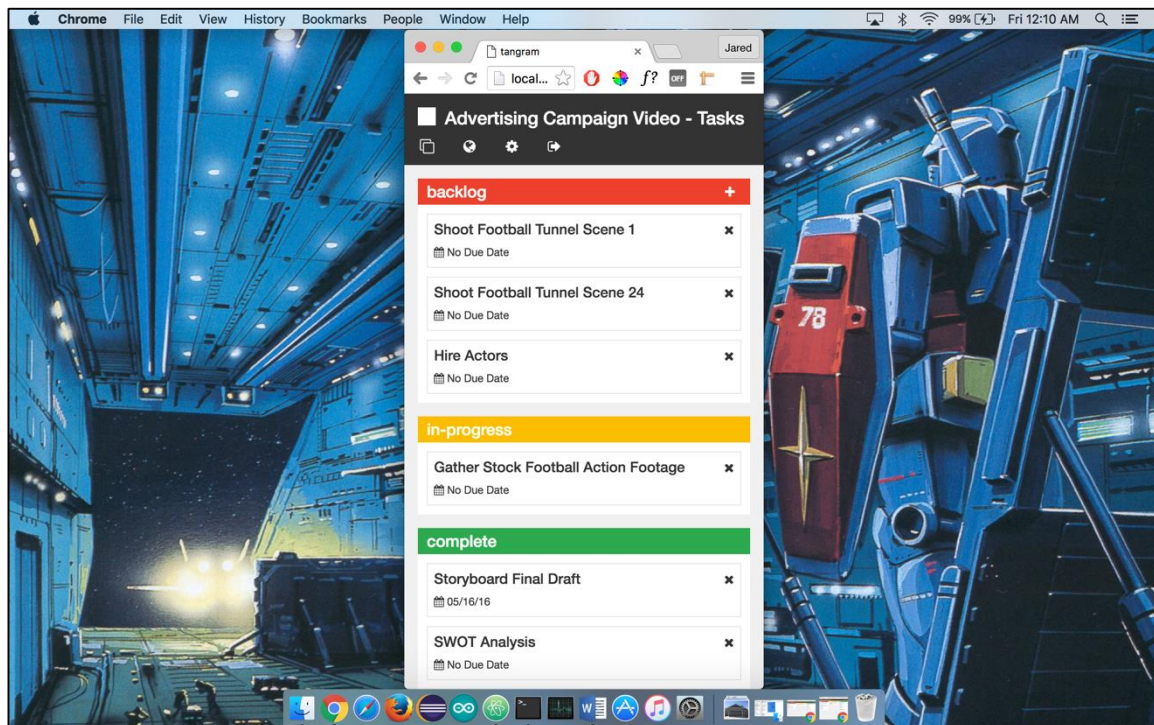
**Figure A1.12** Individual project with inaccurate charts (smartphone)



**Figure A1.13** Project Kanban board (desktop)

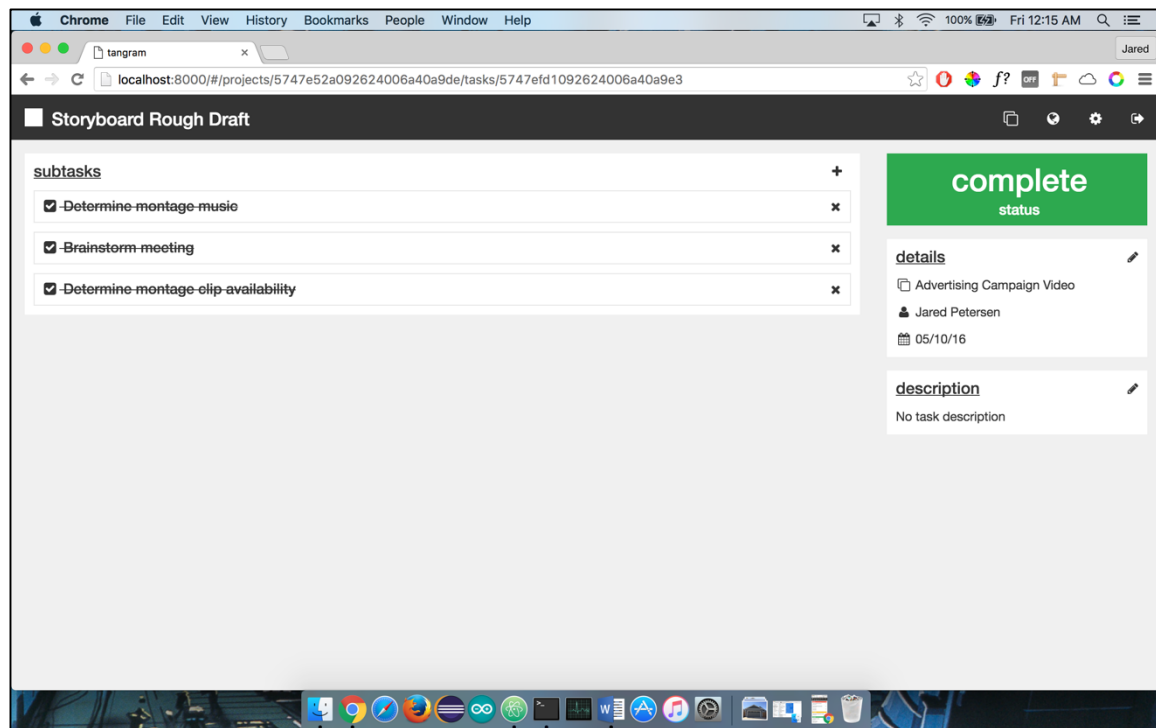


**Figure A1.14** Project Kanban board (tablet)

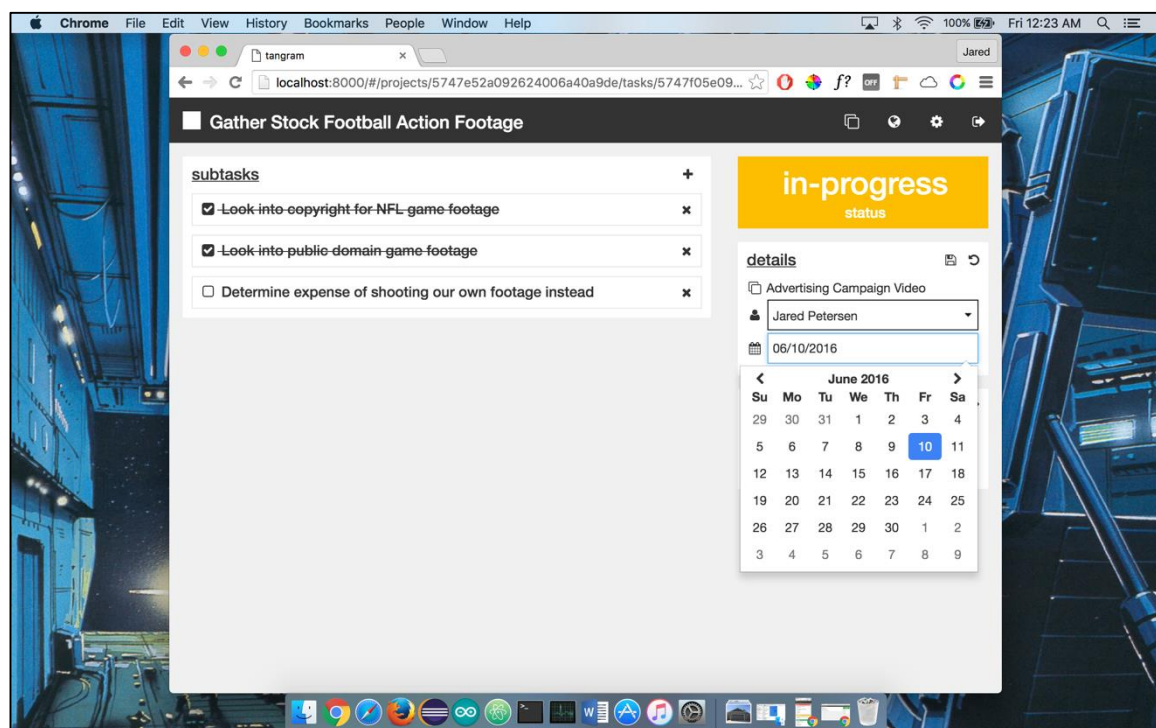


**Figure A1.15** Project Kanban board (smartphone)

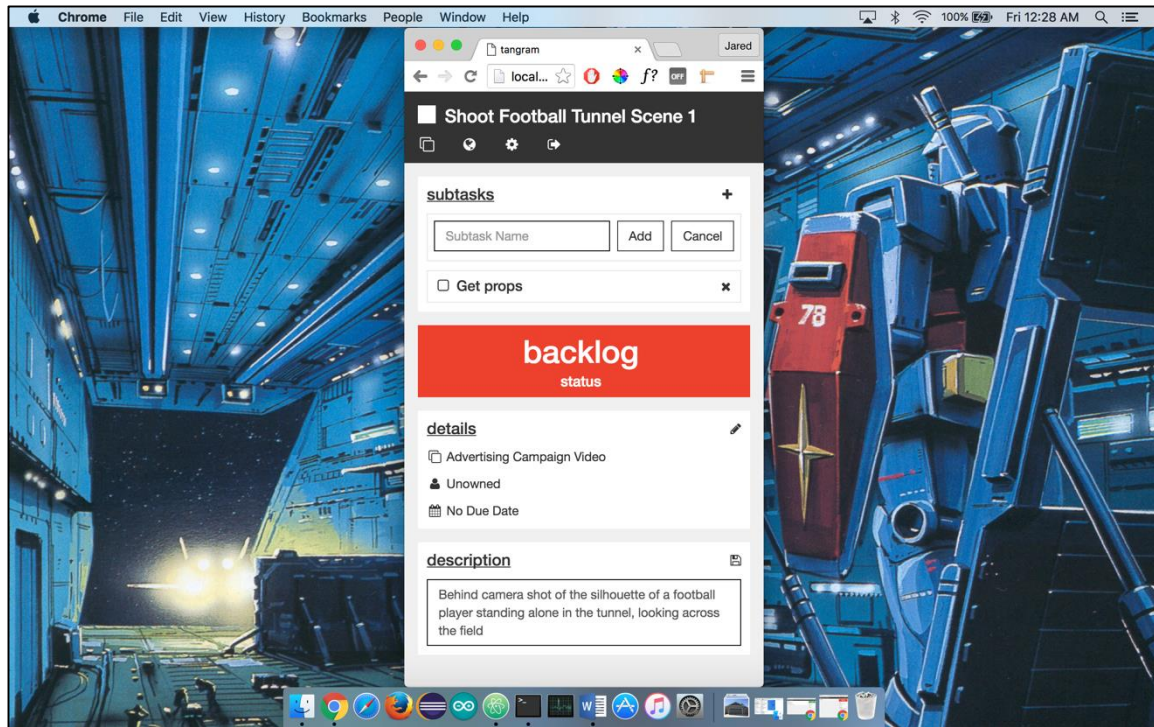




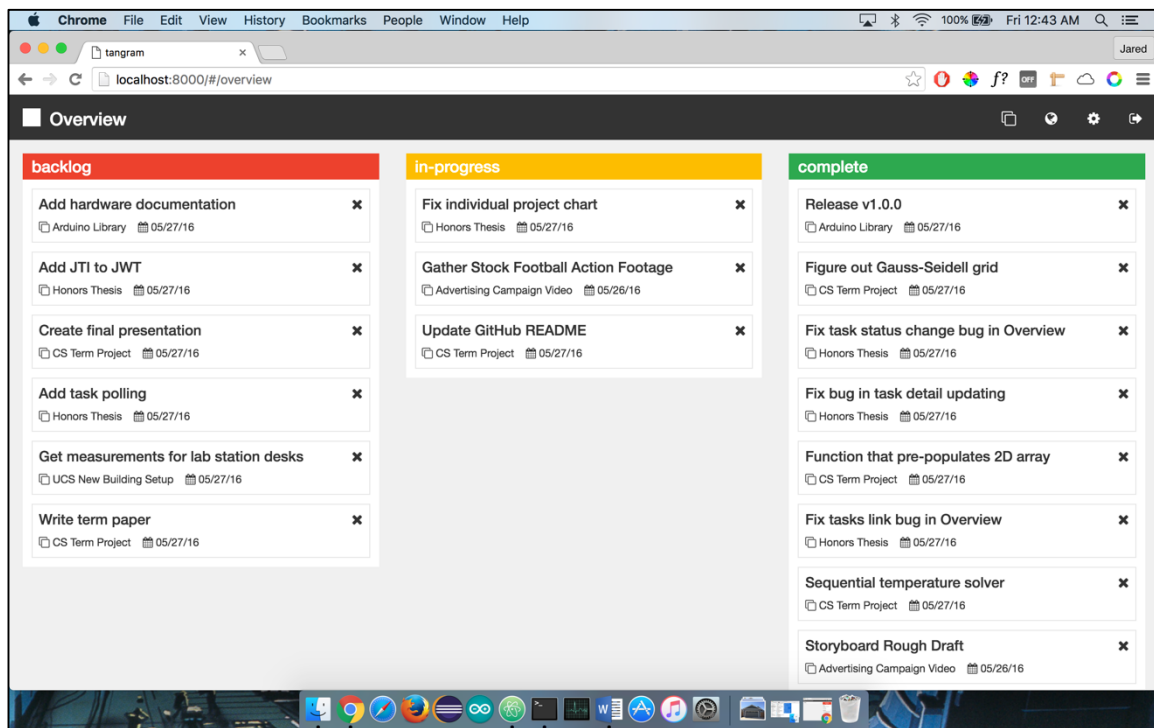
**Figure A1.16** Individual task detail (desktop)



**Figure A1.17** Individual task detail (tablet)

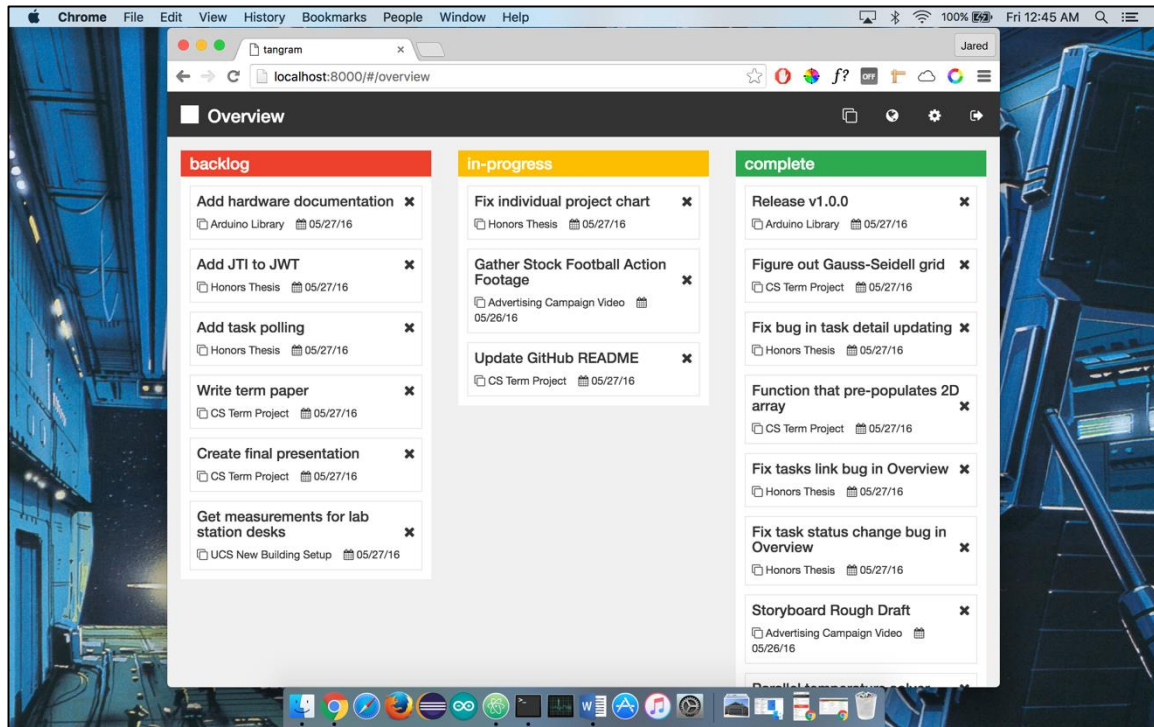


**Figure A1.18** Individual task detail (smartphone)

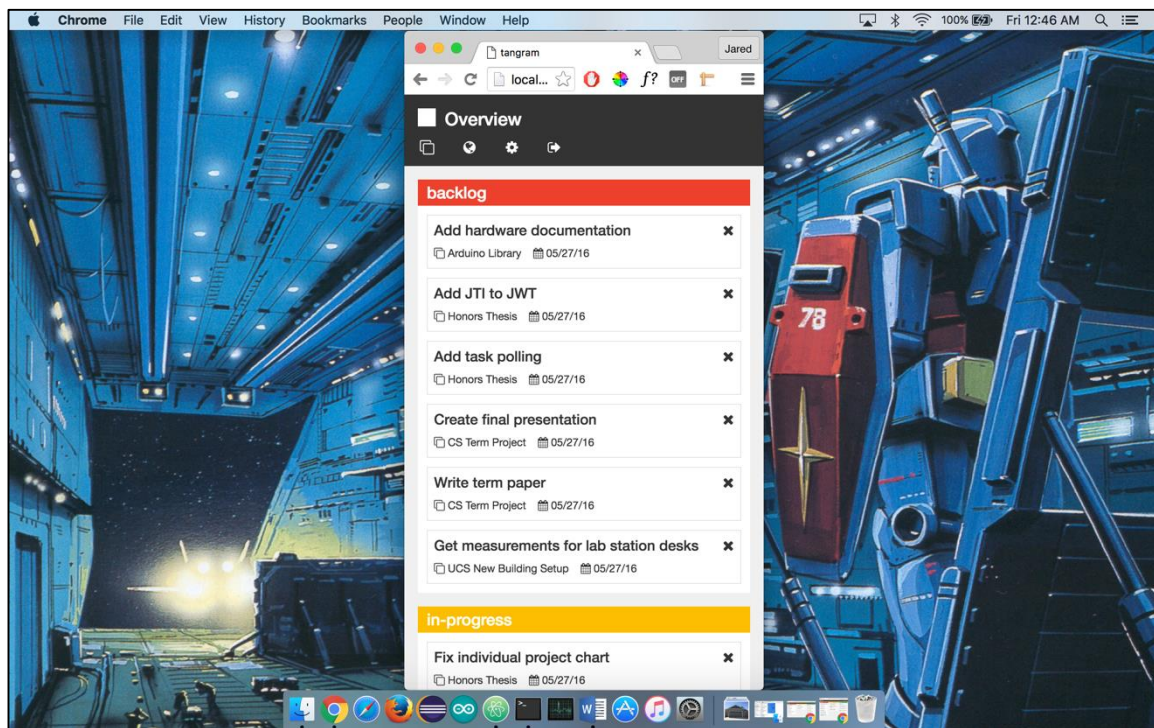


**Figure A1.19** Personal Kanban board/Overview (desktop)





**Figure A1.20** Personal Kanban board/Overview (tablet)



**Figure A1.21** Personal Kanban board/Overview (smartphone)

# Code

All of the code for the backend API and the frontend user interface is located in online GitHub repositories under the MIT license. The backend API code can be accessed via <https://github.com/jaredpetersen/proj-manage-api> and the frontend code can be accessed via <https://github.com/jaredpetersen/proj-manage-front/>.

---

# Bibliography

- [1] T. Ohno, *Toyota Production System: Beyond Large-Scale Production*, Cambridge, MA: Productivity Press, 1988, pp. 25-27.
- [2] D. J. Anderson, *Kanban: Successful Evolutionary Change for Your Technology Business*, Sequim, WA: Blue Hole Press, 2010, pp. 5-9.
- [3] A. Holovaty, J. Kaplan-Moss, *The Definitive Guide to Django: Web Development Done Right*, 2nd ed. Berkeley, CA: Apress, 2009, pp. 7.
- [4] V. Karpov. (2013, Apr. 30). *The MEAN Stack: MongoDB, ExpressJS, AngularJS and Node.js* [Online]. Available:  
<http://blog.mongodb.org/post/49262866911/the-mean-stack-mongodb-expressjs-angularjs-and>
- [5] MongoDB. *Databases and Collections* [Online]. Available:  
<https://docs.mongodb.com/manual/core/databases-and-collections/>
- [6] C. Kvalheim. *Schema Basics* [Online]. Available:  
<http://learnmongodbthehardway.com/schema/chapter2/>
- [7] S. Hess. (2012, Feb. 1). *You Only Wish MongoDB Wasn't Relational* [Online]. Available: [http://seanhess.github.io/2012/02/01/mongodb\\_relational.html](http://seanhess.github.io/2012/02/01/mongodb_relational.html)
- [8] W. Zola. (2015, May 29). *6 Rules of Thumb for MongoDB Schema Design: Part 1* [Online]. Available: <http://blog.mongodb.org/post/87200945828/6-rules-of-thumb-for-mongodb-schema-design-part-1>
- [9] MongoDB. *Data Model Design* [Online]. Available:  
<https://docs.mongodb.com/manual/core/data-model-design/>
- [10] JWT.IO. *JSON Web Tokens* [Online]. Available: <https://jwt.io/>
- [11] JWT.IO. *Introduction to JSON Web Tokens* [Online]. Available:  
<https://jwt.io/introduction/>
- [12] C. Hale. (2010, Jan. 31). *How to Safely Store a Password* [Online]. Available:  
<https://codahale.com/how-to-safely-store-a-password/>
- [13] V. Sahni. *Best Practices for Designing a Pragmatic RESTful API* [Online]. Available: <http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api>

- [14] Linode. (2014, Jul. 24). *Securing Your Server* [Online]. Available: <https://www.linode.com/docs/security/securing-your-server/>
- [15] G. Rodrigues. (2013, Aug. 30). *How to: Allow Node to Bind to Port 80 Without Sudo* [Online]. Available: <https://gist.github.com/firstdoit/6389682>
- [16] O. Lalonde. (2012, Oct. 24). *Don't Run Node.js as Root!* [Online]. Available: <http://syskall.com/dont-run-node-dot-js-as-root/>
- [17] A. Strzelewicz. (2013, May 20). *PM2* [Online]. Available: <https://github.com/Unitech/pm2>
- [18] Bootstrap. *Migration* [Online]. Available: <http://v4-alpha.getbootstrap.com/migration/>
- [19] B. Green. (2015, Dec. 15). *Angular 2 Beta* [Online]. Available: <http://angularjs.blogspot.com/2015/12/angular-2-beta.html>
- [20] Angular. *5 Min Quickstart* [Online]. Available: <https://angular.io/docs/ts/latest/quickstart.html>
- [21] Google. (2015, May 5). *Building for the Next Moment* [Online]. Available: <http://adwords.blogspot.com/2015/05/building-for-next-moment.html>
- [22] L. Abreu. (2014, May 14). *Why and How to Avoid Hamburger Menus* [Online]. Available: <https://lmjabreu.com/post/why-and-how-to-avoid-hamburger-menus/>
- [23] A. L. Turner. (2014). *The History of Flat Design: How Efficiency and Minimalism Turned the Digital World Flat* [Online]. Available: <http://thenextweb.com/dd/2014/03/19/history-flat-design-efficiency-minimalism-made-digital-world-flat/>
- [24] National Eye Institute. *Facts About Color Blindness* [Online]. Available: [https://nei.nih.gov/health/color\\_blindness/facts\\_about](https://nei.nih.gov/health/color_blindness/facts_about)
- [25] A. Bigman. (2013, Apr. 17). *Why All Designers Need to Understand Color Blindness* [Online]. Available: <https://99designs.com/blog/tips/designers-need-to-understand-color-blindness/>
- [26] C. Turnball. (2011, Jul. 22). *Designing For, and As, a Color-Blind Person* [Online]. Available: <http://webdesign.tutsplus.com/articles/designing-for-and-as-a-color-blind-person--webdesign-3408>
- [27] Mountain Goat Software. *Product Owner* [Online]. Available: <https://www.mountaingoatsoftware.com/agile/scrum/product-owner>

- [28] Mountain Goat Software. *ScrumMaster* [Online]. Available:  
<https://www.mountaingoatsoftware.com/agile/scrum/scrummaster>
- [29] Mountain Goat Software. *Scrum* [Online]. Available:  
<https://www.mountaingoatsoftware.com/agile/scrum>
- [30] Atlassian. *A Brief Introduction to Kanban: What Software Makers Can Learn from Japanese Manufacturing* [Online]. Available:  
<https://www.atlassian.com/agile/kanban>
- [31] C. Sevilleja. *The Anatomy of a JSON Web Token* [Online]. Available:  
<https://scotch.io/tutorials/the-anatomy-of-a-json-web-token>
- [32] T. Fowler. *JTI Support #104* [Online]. Available:  
<https://github.com/auth0/node-jsonwebtoken/issues/104>