Slacker News

A minimal Hacker News clone built using Java and the Spring Boot framework

Computer Engineering
California Polytechnic State University
San Luis Obispo, California

Submitted by Calvin Pinson

March 2018

Introduction

Modern web applications incorporate a vast number of technologies and programming paradigms, often presenting an overwhelming landscape of foreign concepts and design patterns. The primary goal of this project is to explore a suite of modern web technologies and concepts through the implementation of a minimalistic Hacker News clone, which is a popular link aggregation site focusing on computer science and entrepreneurship. The primary technologies used in this project are Java and the Spring Boot framework, which provides a simpler interface to the more cumbersome Spring framework, with the goal of allowing for more rapid development.

Background

An exploration of several web programming patterns and technologies is necessary to understand the specific implementations present in the project.

Architecture and Design Patterns

First, we'll take a look at several typically used architectures. In general, there are two primary ones: monolithic architectures, and service-oriented architectures. Monoliths are the most simple, and utilize a single application layer. If one were to write a monolithic web application, the frontend would access data directly, as opposed to accessing an API. Modules are often tightly coupled within a monolith, and scalability can be a problem, but the simplicity of a monolith is often ideal for smaller projects. A service-oriented architecture incorporates several smaller applications communicating over some form of API, as opposed to having a single large application. For example, one might have a customer management service, a cryptography service, and so on and so forth. There's a clear cost in terms of complexity in comparison to the monolithic architecture, and typically, unless the scope of a project exceeds what can be accomplished with a monolith, it's best to stick with the simpler architecture. When necessary, a service-oriented architecture allows significant flexibility, with the capability for each service to be developed and maintained separately, with little knowledge of the functionality of other services. Scaling can also be done more flexibly, with more computing resources being provided to the services which require it. Microservices are one possible implementation of a service-oriented architecture, which in many ways mimics the minimal unix-esque service implementation style. Microservices have absolutely minimal services, as opposed to more traditional service-oriented architectures. Often, architectures that begin as purely monolithic implementations have functionality shorn off into services as they grow, and in general, the boundaries between the architecture patterns are not always clear cut.

There also several organization patterns typically used within the software. One of the most common architectures used is N-layered architecture, or design pattern. An N-layer architecture separates software's logical functionality into several layers, typically a presentation layer, an application layer, a business logic layer, and a data access layer. The term N-tier is often used interchangeably with N-layer architecture, but many regard N-tier as referring to some sort of physical separation; for example, an application with a front-end, an application server, and a database, could be considered 3-tiered. If the business and presentation tiers were not made to be distinct, than a traditional, two-tier client-server model would be the implementation.

Another typical design pattern used in web development (and many other areas) is the model-view-controller, or MVC pattern. Essentially, the model is the central, defining component of the application, which manages any data and logic. The view is a representation of this model, and there can be many views for a single model: for

example, a bar-graph view and a table view of the same data. The controller provides access to the model, accepting user input, validating that input, and then performing operations on the model.

It's important to realize that the MVC pattern, and N-Tier patterns are not exclusive. The UI or presentation layer for an application, for example, may contain the MVC pattern. This UI layer might then communicate with a business logic layer.

At a larger, very general level, there are also distinctions between the front-end and back-end web development. The client portion of the client-server model mentioned earlier is usually considered the front-end. The technologies and concerns of each layer are very different, with the front-end being concerned with HTML, CSS, Javascript, and all of the related technologies and implementations of those languages, while the back-end is concerned with scalability, security, database administration, and many other related concerns. Sometimes, work that is often relegated to the front-end can be accomplished server-side, as with templating engines, which accomplish some of the presentation work by generating parts of the pages that will be served client-side.

Security

There are entire books written on securing web applications, so we'll stick to a basic overview of several relevant topics. One area addressable largely as a whole, is the sanitization of user input. One must assume that any input received by a user is malicious. SQL injection and XSS, whether persistent or reflected, must be addressed through one of several approaches to sanitization. Another aspect of securing web applications is the storage of user data. User passwords for example, should be hashed and stored, while any data that might need to be retrieved in its original form, should be stored encrypted. One other security-related concern is user authentication, and authorization, which put simply, refers to verifying a client's identity, and establishing what they're allowed to access, respectively.

Build Automation

Build automation refers to the automation of common software build processes, including compilation, packaging the compiled binary code, and running any automated tests. Many tools are capable of pulling in dependencies from a remote repository, with no need to store any sources locally. Make would be an example of a build automation tool.

Dependency injection

Dependency injection is a technique in which objects or static methods supply the dependencies of other objects. Here, a dependency is something that is used, and injection is the process of passing the dependency to the object requiring it. The typical alternative here is the use of new or static methods to create the dependency wherever it is needed. Dependency injection is one possible implementation of the inversion of control principle, which is a design principle in which a framework will call the custom portions of a program that have been written by programmers, as opposed to the written code calling the framework. An example of this would be event-driven programming, where custom code handles events passed to it by a framework. Dependency injection adds significant configurability, as one can externalize which objects will passed to other objects as dependencies through separate configuration files that don't require recompilation. This can be very useful when running tests, or performing other tasks which require a specific implementation of a class.

Version Control

Version control simply refers to the tracking and management of changes to some sort of document, or collection of documents. As it relates to computer programming, this refers to tracking of the source code. This is done using a version control system. One example of a version control system is Mercurial. A version control system usually allows to jump between version of the documents as it has changed over time, and to track metadata about those changes. Some allow for the creation of multiple versions which can be created and worked on in parallel. A typical structure for a software project might include several feature branches, and more primary branches for testing, development, and production.

Github

Github is a hosting service for projects using Git as a version control system. Aside from hosting remote repositories for projects alone, Github offers additional functionality in the form of wikis, bug tracking, feature requests, and many other features.

Continuous Integration

Continuous integration refers to merging all development work to a common trunk (a development branch in version control, for example) frequently, often several times a day. The goal of continuous integration is to avoid integration problems that occur when the work of multiple developers becomes increasing out of sync. Often projects that practice continuous integration, include build automation and automatic deployment.

Continuous Deployment

Continuous deployment refers to the practice of automatically deploying every change to production. This requires continuous delivery, which is an approach to software engineering in which teams produce software in very short cycles, so that the software can be released at any time, essentially releasing incremental updates.

Travis CI

Travis CI is a hosted continuous integration service that is capable of building and testing software projects hosted on Github. Users can specify build information in a YAML (a markup language) formatted text file placed in the root of the project's directory, named .travis.yml. This file specifies information about the the programming language used, dependencies, and other necessary build parameters. Travis can be configured to pull from Github once certain conditions have been met, for example, whenever a pull request is submitted, or upon every commit.

Heroku

Heroku is a platform as a service for the deployment of web applications. It supports several languages and database implementations. A customer's applications are ran in virtual containers called Dynos, from code which is typically deployed through Git. One of the most important aspects of Heroku is that nearly all server management and configuration are obfuscated from the client.

Sentry

Sentry is an error monitoring tool for tracking and debugging production errors. Data is collected, and viewable within a dashboard on Sentry's website. This data potentially allows for insightful analysis of the errors being

generated. Though there are a glut of error-tracking tools on the market, Sentry is convenient due to its free tier, which allows for up to 5000 errors per day at no cost.

Spring Framework

The Spring Framework is an open source application framework and inversion of control container built for the Java platform. The Spring Framework offers a number of modules that offer different core functionalities, including authentication and authorization through the Spring Security sub-project, data access through Java Database Connectivity, servlet-based web services, and many others.

Spring Boot

Spring Boot is a simplified framework built on top of Spring, which offers very strong default configurations, removing the need to do much of the typical configuration work in Spring. Essentially, Spring Boot allows for a fairly rapid path to developing and deploying production grade Spring-based applications.

JPA

The JPA, or Java Persistence API, is an API specification for managing interactions between Java objects and relational databases, including the accessing and storing of those objects. JPA is only a specification for object relational mapping, not an actual implementation. An example of a specific implementation of JPA would be Hibernate.

Spring Data JPA

Spring Data JPA is an additional layer of abstraction on top of JPA that aims to simplify implementation of JPA.

ORM

ORM, or object-relational mapping, is a technique for converting data between different forms of representation between different type systems; for example, between an Java object and a row in a database table.

Annotations

Java annotations provide metadata about the source code they're annotating. These annotations actually remain in class files generated by the compiler, and can be read at run-time by the JVM. They can be used to provide custom behavior on top the of the default annotations included in Java, like @Override.

PostgreSQL

PostgreSQL, commonly abbreviated to Postgres, is a popular ORDBMS. Heroku includes built-in support for Postgres, which is why it's been chosen here as opposed to any other form of database.

Thymeleaf

Thymeleaf is a popular Java templating engine. Templating engines take templates, which is often in some form of markup language, alongside some form of data model, to generate documents. This can be in the form of web pages, or other applications. Often this allows one to use loops, variables, and other aspects typical to programming to generate the front-end of a document. This can save a significant amount of labor.

Description

As mentioned previously, this project is a web application built using Java and Spring Boot in imitation of the popular link aggregator, Hacker News. The chosen infrastructure is that of a monolith, which is sensible given the simplicity and limited scope of this project. The software itself is built using both the MVC design pattern mentioned earlier, and the N-tier pattern (specifically, N-level, as the software itself is divided into different logical strata, while being hosted on a single device). Generally, the project follows the popular 3-tier model, having controllers, services, and repositories where data access occurs. There also models and configuration files. The general project structure can be seen in figure 1.

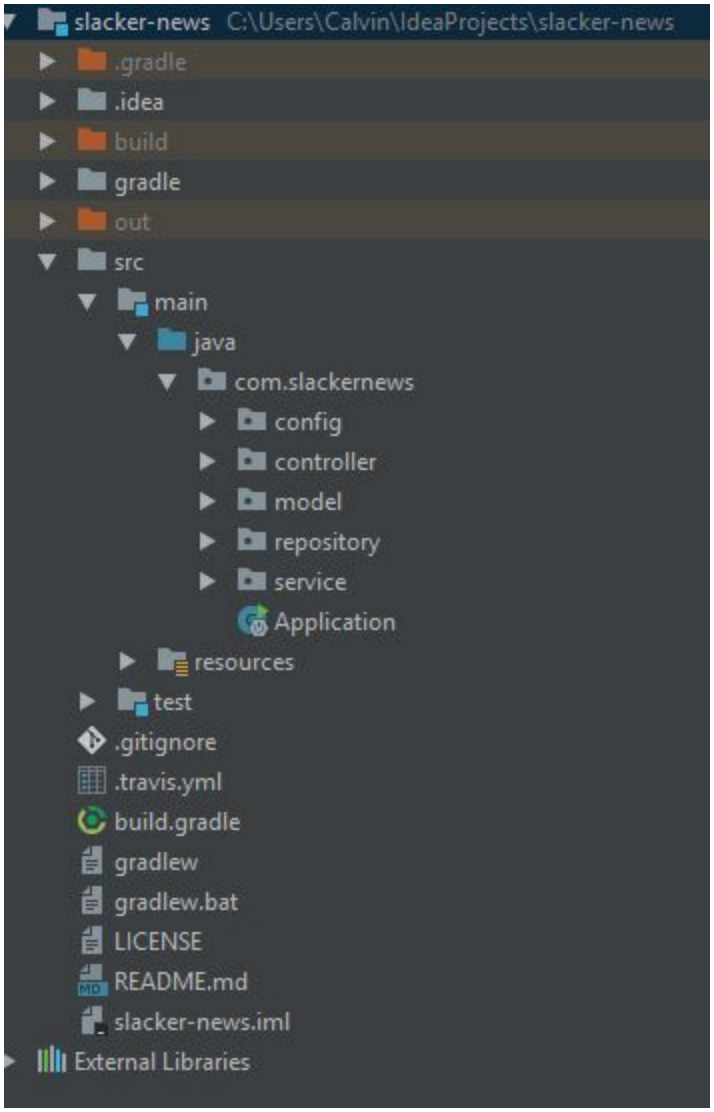

Figure 1: Project Structure

build.gradle

We'll first take a look at the project dependencies, which are pulled in using Gradle. This information, alongside other build information, is stored in the build.gradle file. Gradle uses this information to build the project. The other files, gradlew and gradlew.bat, are parts of the Gradle Wrapper, which is essentially a script which invokes a version of Gradle. The wrapper can install gradle, and is tied to a specific gradle version, which

can increase build reliability, and means that you don't need to manually install gradle on whatever system the build is being run on. The content of the gradle file can be seen in Figure 2.

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:1.5.7.RELEASE")
    }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'idea'
apply plugin: 'org.springframework.boot'

jar {
    baseName = 'gs-slacker-news'
    version =  '0.1.0'
}

repositories {
    mavenCentral()
}

sourceCompatibility = 1.8
targetCompatibility = 1.8

dependencies {
    compile("org.springframework.boot:spring-boot-starter-web")

    // JPA Data (We are going to use Repositories, Entities, Hibernate, etc...)
    compile 'org.springframework.boot:spring-boot-starter-data-jpa'

    compile("org.postgresql:postgresql")

    compile("org.springframework.boot:spring-boot-starter-security")

    //The templating engine
    compile("org.springframework.boot:spring-boot-starter-thymeleaf")

    compile("org.thymeleaf.extras:thymeleaf-extras-springsecurity4")

    //Add sentry integration for error aggregation
    compile('io.sentry:sentry:1.6.6')

    testCompile('org.springframework.boot:spring-boot-starter-test')
}
```

Figure 2: build.gradle

The most relevant aspects shown in the gradle file are the remote repository, Maven Central, which is where many of the listed dependencies will be downloaded from. Among the project dependencies are Postgresql, which is the database variety that will be connected with, the Spring Boot Starter Data JPA package, which as mentioned earlier, is part of the simplified JPA API unique to spring, which we'll be using to store our Java objects as tuples in a database, Spring Boot Starter Security, which will be used to provide security configuration, Thymeleaf, which is the templating engine which will be used to render many of the pages served to users, some Thymeleaf extras for additional functionality, and Sentry, for error logging.

Travis.yml

The other primary configuration file in the root directory is the .travis.yml file, which instructs Travis CI on how the project should be built, as well as including an encrypted token which allows Travis to connect to Slack and push notifications about the build and test process.

Models

One of the most defining characteristics of the project are the models. There are three primary models used: Comment, Post, and User. Each of the models is somehow related to the other two; for example, a user can post both comments, so multiple comments and posts can belong to a single user.

A user is defined by several characteristics, including an email, username, password hash, the set of comments they've made, and the set of posts they've made. These are stored as private variables within the User class. Simply by adding the Entity annotation to a class, JPA will understand that this class is intended to be mapped to a database. In the user class shown the name is overridden to "user_entity," as opposed to the default class name being used, because "User" is a reserved keyword in Postgres. The annotations present on the getters in the user class similarly convey information to JPA about the variables. Most of the annotations are self-explanatory, but the annotations on the related model variables, posts, and comments, are worth examining. The "mappedBy" annotation will refer to how the other models, Post, and Comment, reference the User model. For example, in the Post class, there is a private User variable which is named "Poster." The cascade type shown refers to how changes in the database propagate (cascade), meaning the cascade type defines what will happen to related models when one is deleted - for example, is a User is deleted, their comments will also be removed. The "OneToMany" relationship describes a relationship exactly as it sounds - for every one User, there can be multiples of posts, and multiples of comments. The opposite is seen in the Comment and Post classes.

```
package com.slackernews.model;

import javax.persistence.*;
import java.util.Set;

// Need a custom name, because you can't have a table named 'user' in postgres
@Entity(name="user_entity")
public class User {
   private Integer id;
   private String email;
   private String username;
   private String passwordHash;
```

```
    private Set<Post> posts;
    private Set<Comment> comments;

    // Necessary for JPA
    protected User() {};

    public User(String username, String email, String passwordHash) {
        this.username = username;
        this.email = email;
        this.passwordHash = passwordHash;
    }

    @Override
    public String toString() {
        return "USER";
    }

    //region Getters and Setters
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(nullable = false, updatable = false)
    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    @Column(nullable = false)
    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    @Column(nullable = false, unique = true)
    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    @Column(nullable = false)
    public String getPasswordHash() {
        return passwordHash;
    }

    public void setPasswordHash(String password) {
        this.passwordHash = password;
    }
```

```
    @OneToMany(mappedBy = "poster", cascade = CascadeType.ALL)
    public Set<Post> getPosts() {
        return posts;
    }

    public void setPosts(Set<Post> posts) {
        this.posts = posts;
    }

    @OneToMany(mappedBy = "commenter", cascade = CascadeType.ALL)
    public Set<Comment> getComments() {
        return comments;
    }

    public void setComments(Set<Comment> comments) {
        this.comments = comments;
    }

    //endregion

}
```

Figure 3: User Class

One of the other patterns of class contained with the other models is that of the CreationForm. There is a CommentCreationForm class, a PostCreationForm class, and a UserCreationForm class. The classes are used to pass any necessary variables for creating a model from controllers to the service layer. They are in effect modeled after the data access object design pattern, or DAO pattern, which attempt to abstract away some of the implementation details of the underlying service.

```
package com.slackernews.model;

import org.hibernate.validator.constraints.NotEmpty;

public class UserCreationForm {

    @NotEmpty
    private String username = "";

    @NotEmpty
    private String email = "";

    @NotEmpty
    private String password = "";

    @NotEmpty
    private String passwordRepeated = "";


    //Getters and Setters
    //Helper Methods
```

```
}
```

Figure 4: User Creation Form

The final model pattern present is the Validator class type. In this project, there is only a UserCreationFormValidator, though there could easily be a justification for comment and post validators. The UserCreationFormValidator checks each field in the UserCreationForm for validity, according to certain conditions; for example, making sure the password and repeated password match, and that the username and email are unique. The component annotation signals to Spring that this class is a generic Spring component. This allows the class to be used in the validator pattern that is an inherent part of Spring. The overridden supports method is present for the same reason. One very interesting this to note is the @Autowired annotation on the constructor for the class, which takes in a UserService object. This is an example of the dependency injection that Spring is capable of performing. Rather than creating its own userService, or referencing a static implementation, the framework itself is capable of recognizing the dependency, and providing it as necessary.

```java
@Component
public class UserCreationFormValidator implements Validator {

    private final UserService userService;

    @Autowired
    public UserCreationFormValidator(UserService userService) {
        this.userService = userService;
    }

    @Override
    public boolean supports(Class<?> clazz) {
        return clazz.equals(UserCreationForm.class);
    }

    @Override
    public void validate(Object target, Errors errors) {
        UserCreationForm form = (UserCreationForm) target;
        validateUsername(errors, form);
        validatePasswords(errors, form);
        validateEmail(errors, form);
    }

    private void validatePasswords(Errors errors, UserCreationForm form) {
        if (!form.getPassword().equals(form.getPasswordRepeated())) {
            errors.reject("password.no_match", "Passwords do not match");
        }
    }

    private void validateUsername(Errors errors, UserCreationForm form) {
        if (userService.getUserByUsername(form.getUsername()).isPresent()) {
            errors.reject("username.exists", "Username is already taken");
        }
    }

    private void validateEmail(Errors errors, UserCreationForm form) {
        if (userService.getUserByEmail(form.getEmail()).isPresent()) {
            errors.reject("email.exists", "User with this email already exists");
        }
    }
}
```

Figure 5: User Creation Form Validator

Services

The service layer, as mentioned in the background section, is a layer that handles transforming and manipulating data as it passes between the presentation and data layers. The data layer in this case will be the repository, and the presentation layer includes the controllers and the views that are passed to clients. There are three primary services being used: the CommentService, PostService, and UserService, as consistent with the

models that have been examined so far. These services are defined as interfaces, with the goal of being able to define implementation flexibly; however, given the limited scope of this project, I've only defined one implementation for each contract. There are also several services that have gone unused. In the UserService, there are several methods with return types of Optional<User>; this is a functional programming pattern, where Optional is a container intended to hold an object that isn't guaranteed to exist. It's very possible to look up a user by an invalid value, and the Optional type helps to eliminate some boilerplate. The way this type of value is handled will be seen in the UserController. The create method returns a User every time, as by the time the execution has reached this portion of the service, the data in the UserCreationForm has already been validated, and the User is guaranteed to be unique. The "I" prefix on the interface classes is used to indicate that the class is an interface, to distinguish the interface from the concrete implementation.

```
public interface IUserService {
   Optional<User> getUserById(int id);

   Optional<User> getUserByEmail(String email);

   Optional<User> getUserByUsername(String username);

   User create(UserCreationForm form);
}
```

Figure 6: IUserService

The Service annotation is used to indicate that this class performs service tasks, but there is no functionality provided by the Spring Framework due to this annotation. Its use is more akin to a best practice to delineate a separation of concerns pattern. Here the Autowired annotation can be seen once more, this time for the injection of a UserRepository, which is the data access layer of this project. Most methods are a direct passthrough to the repository, except for the create method. In the create method, relevant variables are pulled from the form, and password hash created using the BCryptPassword encoder, before a user is created and stored using the UserRepository.

```
@Service
public class UserService implements IUserService{

   private final IUserRepository userRepository;
   private static final Logger log = LoggerFactory.getLogger(UserService.class);


   @Autowired
   public UserService(IUserRepository userRepository) {
       this.userRepository = userRepository;
   }

   @Override
   public Optional<User> getUserById(int id) {
       return Optional.ofNullable(userRepository.findOne(id));
   }
```

```
    @Override
    public Optional<User> getUserByUsername(String username) {
        return userRepository.findOneByUsername(username);
    }

    @Override
    public Optional<User> getUserByEmail(String email) {
        return userRepository.findOneByEmail(email);
    }

    @Override
    public User create(UserCreationForm form) {
        log.info("Creating user");
        String username = form.getUsername();
        String email = form.getEmail();
        String passwordHash = new BCryptPasswordEncoder().encode(form.getPassword());

        User user = new User(username, email, passwordHash);

        return userRepository.save(user);
    }
}
```

Figure 7: UserService

Repositories

The repository layer is where any data access occurs. There is a repository for each model, User, Comment, and Post. Interestingly, no concrete implementations are necessary. All that is needed is an interface which extends JpaRepository, and simply defining the methods one would like. JpaRepository provides a number of generic methods that allow for database interactions, which save an enormous amount of work.

```
public interface IUserRepository extends JpaRepository<User, Integer> {
    Optional<User> findOneByEmail(String email);
    Optional<User> findOneByUsername(String username);
}
```

Figure 8: IUserRepository

Controllers

The controllers are part of the presentation layer, and are the endpoints with which a user will interact. The controller annotation indicates that the class is a controller to the Spring MVC framework. The DispatcherServlet class (implemented and configured by default in Spring Boot) scans for annotated classes, and detects the RequestMapping annotations, and uses those to determine where to dispatch HTTP requests. The RequestMapping annotations convey information about the URI and other aspects of the HTTP request that will correspond to the method; for example, the getUserCreationPage method in the UserController seen in Figure 9 is mapped to "/user/create," which an extension to the base URL, determined at runtime. Run locally, the URI to access this endpoint would be "localhost:8080/user/create," for example. One can also specify the type of HTTP request, such as GET, POST, or PUT. One of the common return types seen in controllers is

ModelAndView. The model and view refer to the concepts in the model view controller design pattern explained in the background section. With this approach, the code of the method can be more succinct

```
@Controller
public class UserController {

   private final IUserService userService;
   private final UserCreationFormValidator userCreateFormValidator;

   @Autowired
   public UserController(UserService userService, UserCreationFormValidator userCreationFormValidator) {
       this.userService = userService;
       this.userCreateFormValidator = userCreationFormValidator;
   }

   @InitBinder("form")
   public void initBinder(WebDataBinder binder) {
       binder.addValidators(userCreateFormValidator);
   }

   @RequestMapping("/user/{id}")
   public ModelAndView getUserPage(@PathVariable int id) {
       return new ModelAndView("user", "user", userService.getUserById(id)
               .orElseThrow(() -> new NoSuchElementException(String.format("User=%s not found", id))));
   }

   @RequestMapping(value = "/user/create", method = RequestMethod.GET)
   public ModelAndView getUserCreationPage() {
       return new ModelAndView("user_create", "form", new UserCreationForm());
   }

   @RequestMapping(value = "/user/create", method = RequestMethod.POST)
   public String handleUserCreationForm(@Valid @ModelAttribute("form") UserCreationForm form, BindingResult
bindingResult) {
       if (bindingResult.hasErrors()) {
           return "user_create";
       }
       try {
           userService.create(form);
       } catch (DataIntegrityViolationException e) { //TODO: Add logging for the exception
           bindingResult.reject("username.exists", "That username is already taken");
           return "user_create";
       }
       return "redirect:/";//TODO: custom redirect?
   }

}
```

Figure 9: User Controller

Configuration and Security

Much of the configuration for a Spring Boot application can be done in code - though there's often very little configuration required. The primary configuration within the project is all security-related. There's a small portion of Thymeleaf configuration, but that too is security-related.

A large portion of the necessary security configuration is specifying information about the process of authentication, logging out, and the authorization of various URLs and paths. The antMatchers pattern seen refers to Ant-style path patterns, Ant being the old Apache build system. This pattern is what is used to permit various paths, which include a login page, a login failure redirect, a login success redirect, logout, and the login parameters being selected. A CurrentUserDetails service is injected, and used within the configure method alongside BCrypt, which was selected as a viable hashing algorithm for password storage. The CurrentUserDetailsService essentially wraps the UserService with some additional functionality provided from Spring Security. Originally, the CurrentUser model was intended only as a data access object to obscure some of the unnecessary details of the User class, but with so little data in the User, it's instead been included as a private member of the class. Spring Security enables a number of secure defaults, including protection against cross site scripting and cross-site request forgery. Authentication and authorization are also implemented securely by default, removing the need to do little other than indicating where the appropriate URLs are. There's not any explicit protection against SQL injection that must be done either, as all queries are performed using the built-in parameterized queries of JPA, as opposed to building SQL statements explicitly using user defined variables.

```
@Configuration
@Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)
class SecurityConfig extends WebSecurityConfigurerAdapter {

   private UserDetailsService userDetailsService;

   @Autowired
   public SecurityConfig(CurrentUserDetailsService userDetailsService) {
       this.userDetailsService = userDetailsService;
   }

   @Override
   protected void configure(HttpSecurity http) throws Exception {
       http.authorizeRequests()
               .antMatchers("/", "/user/create", "/login","/post/**").permitAll()
               .antMatchers("/vote/**", "/comment/submit").authenticated()
               .antMatchers(HttpMethod.POST, "/submit").authenticated()
               .and()
               .formLogin()
               .loginPage("/login")
               .failureUrl("/login?error")
               .defaultSuccessUrl("/", true)
               .usernameParameter("username")
               .passwordParameter("password")
               .permitAll()
               .and()
               .logout()
               .logoutUrl("/logout")
               .logoutSuccessUrl("/")
```

```
                .permitAll();
    }

    @Override
    public void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth
                .userDetailsService(userDetailsService)
                .passwordEncoder(new BCryptPasswordEncoder());
    }

}
```

Figure 10: Security Configuration

Resources

Generally, anything contained in the resources folder is regarded as static. This typically includes front-end assets, like images, CSS, and Javascript, and the thymeleaf templates that are used to render the pages that are served. These pages are first processed server-side, and are used to generate pure html pages that are then provided to clients. Thymleaf tags, seen throughout as “th” can be used to provide a significant amount of functionality that could be cumbersome to implement using traditional web technologies alone. If there were a large number of clients being served, it would likely be better to pass off the rendering responsibilities to the clients, to reduce server load. This would mean sending more necessary data asynchronously, likely in the form of JSON. Given the small scale of the project, I felt it would be appropriate to make use of Thymeleaf. Besides providing traditional functionality, as seen with setting the “hidden” property, or specifying actions, Thymeleaf allows you to work with Java objects to some degree, which can drastically simplify certain aspects of front-end development. This page populates a long a list of posts, which can be done with a for-each loop, and then accesses methods of those posts to fill-out the necessary information on the front end.

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="UTF-8"/>
    <title>Slacker News</title>
    <meta name="referrer" content="origin"></meta>
    <meta name="viewport" content="width=device-width, initial-scale=1.0"></meta>
    <link rel="stylesheet" type="text/css" href="/css/home.css"></link>
    <link rel="shortcut icon" href="/images/favicon.ico" type="image/x-icon"></link>
</head>
<body>
<center>
<form name="logoutForm" th:action="@{/logout}" method="post" th:hidden="true">
    <input hidden="true" type="submit" value="Sign Out"/>
</form>
<table id="hnmain" border="0" cellpadding="0" cellspacing="0" width="85%" bgcolor="#f6f6ef">
    <tbody>
        <tr>
            <td bgcolor="#ff6600">
                <table border="0" cellpadding="0" cellspacing="0" width="100%" style="padding:2px">
                    <tbody>
```

```
                    <tr>
                        <td style="width:18px;padding-right:4px">
                            <a href="https://news.ycombinator.com">
                                <img src="/images/s18.png" width="18" height="18"
                                     style="border:1px white solid;"/>
                            </a>
                        </td>
                        <td style="line-height:12pt; height:10px">
                            <span class="pagetop">
                                <b class="hnname">
                                    <a th:href="@{/}">Slacker News</a>
                                </b>
                                <a href="newest">new</a>
                                 |
                                <a href="newcomments">comments</a>
                                 |
                                <a th:href="@{/submit}">submit</a>
                            </span>
                        </td>
                        <td style="text-align:right; padding-right:4px">
                            <span sec:authorize="isAuthenticated()" class="pagetop">
                                <a th:href="@{/user/details}"
th:text="${#authentication.getPrincipal().getUser().getUsername()}"></a>
                                |
                                <a href="javascript: document.logoutForm.submit()"
role="menuitem">logout</a>
                            </span>
                            <span sec:authorize="isAnonymous()" class="pagetop">
                                <a th:href="@{/login}">login</a>
                            </span>
                        </td>
                    </tr>
                </tbody>
            </table>
        </td>
    </tr>
    <tr style="height:10px"></tr>
    <tr>
        <td>
            <table border="0" cellpadding="0" cellspacing="0" class="itemList">
                <tbody>
                    <th:block th:each="post : ${posts}">
                        <tr class="athing" th:id="${post.getId()}">
                            <td align="right" valign="top" class="title">
                                <span class="rank"></span>
                            </td>
                            <td valign="top" class="voteLinks">
                                <center>
                                    <a th:id="${post.getId()}"
th:href="@{/vote/post(id=${post.getId()})}">
                                        <div class="votearrow" title="upvote"></div>
                                    </a>
                                </center>
                            </td>
                            <td class="title">
```

```
                                    <a th:href="${post.getURL()}" class="storyLink"
th:text="${post.getTitle()}"></a>
                                </td>
                            </tr>
                            <tr>
                                <td colspan="2"></td>
                                <td class="subtext">
                                    <span class="score" th:id="${'score_' + post.getId()}"
th:text="${post.getPoints() + ' points'}"></span>
                                     by
                                    <a href="" class="hnuser" th:text="${post.getPoster()}"></a>
                                    <span class="age">
                                        <a href="" th:text="${'Posted at ' + post.getPostDate()}"></a>
                                    </span>
                                     |
                                    <a th:href="@{/post/} + ${post.getId()}"
th:text="${post.getComments().size() + ' Comments'}"></a>
                                </td>
                            </tr>
                            <tr class="spacer" style="height:5px"></tr>
                        </th:block>
                    </tbody>
                </table>
            </td>
        </tr>

    </tbody>
</table>
</center>
</body>
<script type='text/javascript' src='/javascript/home.js'></script>
</html>
```

Figure 11: home.html

Application

This class is very simple. Its only purpose is to launch the application itself. The @SpringBootApplication annotation is a convenience annotation that adds several annotations, including @Configuration, which tags this class as a source of bean definitions, @EnableAutoConfiguration, which configures Spring Boot to automatically add beans based on the path, existing beans, and other properties, and @ComponentScan, which instructs Spring to look for components and configurations in the same package as this file. Normally @EnableWebMvc would be included too, but Spring Boot detects that spring-webmvc is on the classpath, and automatically configures the application as a web application and sets of the necessary DispatcherServlet.

```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class);
    }
}
```

Figure 12: Application.java

Testing

Spring boot offers several built in testing features. The annotations here are worth examining: @SpringBootTest instructs Spring Boot to find a main configuration class to start a Spring Application context (here, the main configuration class would be the one annotated with @SpringBootApplication, which is the Application.java class which was previously examined). @RunsWith is a JUnit annotation which specifies the class to be used to run the tests in the calling class. The @Test annotation is also a JUnit annotation, which defines behavior regarding how the method should be treated, with the ability to specify information like expecting an exception to occur. These annotations also allow the test to be run using commands with Gradle or other build automation systems. The test listed only checks to see if the Spring Application Context has been created, which is the central configuration interface for the application.

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class ApplicationTest {

   @Test
   public void contextLoads() throws Exception {
   }
}
```

Figure 13: ApplicationTest.java

Much of the same information can be seen in the ControllerSmokeTest class. This test ensures that the injected HomeController has been created.

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class ControllerSmokeTest {

   @Autowired
   private HomeController controller;

   @Test
   public void contextLoads() throws Exception {
       assertThat(controller).isNotNull();
   }
}
```

Figure 14: ControllerSmokeTest.Java

Deployment

The basic continuous deployment pipeline for the project uses a combination of Github, Travis CI, Heroku, Sentry, and Slack. Code is deployed to Github using a free public repository. This code is then tested on Travis CI - if all tests pass, the code is then deployed to Heroku, which includes a Postgres database. Sentry is used to log any errors (admittedly, I've enabled this integration without making use of it). All notifications about the build and deployment process are pushed to Slack. All of this is done using the free tier of each of these products, and if necessary, moving up the paid tier would provide a continuous deployment pipeline robust enough for a number of mid-size projects.

Evaluation

This project succeeds in terms of being a very bare-bones implementation of a Hacker News clone, though it lacks significantly in polish. The final product is not feature rich, and lacks some of what I would consider necessary to a be a true minimal clone of Hacker News, including nested comment trees, and the ability to manage your posts as a user, including deletion. I think additional work customizing the front-end of the project, while it would move the project away from being a visual clone of Hacker News, would also be a worthwhile venture, as the set of skills and technologies used are very different. The test coverage on the project is also very poor, and I'd have liked to have a more comprehensive suite of unit and integration tests throughout the project. I also would have liked to have significantly more data logging and analytics made available, perhaps visualed through some form of dashboard.

Conclusions

In general, this project provided an excellent opportunity to familiarize myself with many of the technologies I expect to encounter throughout my career. Though development could often be frustratingly slow-going when encountering obtuse errors from aspects of the project I was most unfamiliar with, troubleshooting those errors pose learning opportunities in themselves. Much of this project consisted of establishing a reasonable scaffold for future work, and future expansions on whatever has been developed should move along nicely using and expanding upon existing modules. The first areas for improvement would be the post karma mechanic and the display order for posts, which currently lack any decay over time; this means top-rated posts will always be at the top, no matter how old. There should also be an asynchronous API for upvoting comments and posts, a comment reply mechanic that allows for nested comment trees, the ability to view a user's posts and comments from a user profile page, an administration mechanic that allows the removal of posts, comments, and banning users, a robust password reset mechanism using an email password reset link, a cryptography service that allows for KEK and DEK rotation, a password complexity checker that ensures a minimum level of password security, authentication using OAUTH, some form of cache-ing for results, high-availability provided through some content delivery network, and many other features. With this project, there are a huge number of potential expansions and features. It wouldn't be unreasonable to pivot to different platforms for deployment, or otherwise change the current deployment pipeline. Expanding to a workflow that includes multiple environments for staging, development, testing, and production, and associated branches within version control would also make for interesting changes on the devops side of development. Moving from free tools to a

self-hosted server, with appropriate backup practices, practical logging, and otherwise creating a robust IT environment would also make for interesting expansions. Again, this project excels in versatility and expandability, and every aspect of enterprise web development is an accessible option.

References:

BCrypt
https://www.usenix.org/legacy/events/usenix99/provos/provos_html/node1.html

Git
https://git-scm.com/doc

Github
https://help.github.com/

Heroku
https://devcenter.heroku.com/categories/reference

Java Documentation
https://docs.oracle.com/javase/8/

PostgreSQL
https://www.postgresql.org/docs/

Sentry
https://docs.sentry.io/

Software Architecture and Design
https://msdn.microsoft.com/en-us/library/ee658093.aspx

Spring Framework
https://spring.io/docs

Spring Boot Documentation
https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/

Spring Data JPA
https://docs.spring.io/spring-data/jpa/docs/current/reference/html/

Thymeleaf Documentation
https://www.thymeleaf.org/documentation.html

Travis CI

https://docs.travis-ci.com/

Web Security
https://www.owasp.org/index.php/Main_Page