Alshayun: A mobile content-delivery application

Jacob Chappell

March 29, 2019

Contents

List of Figures

Chapter 1

Introduction

1.1 What is Alshayun?

Alshayun is a mobile application for delivering articles consisting of rich text and interactive **applet**s to **reader**s. Written in the portable **Ionic** framework, **Alshayun** is capable of running on the Web, Android devices, and Apple iOS devices. However, **Alshayun** has currently only been tested on a Samsung Galaxy S7 device running Android and a Google Pixel emulator.

Alshayun is designed with three actors in mind: the **content author**, the **reader**, and the **developer**. The **content author** is anyone who has anything they would like to write about, potentially making use of interactive and embedded **applet**s as assistive visual aids. The **reader** is anyone who would like to read what one or more **content author**s have to say. The **developer** is the one capable of developing **applet**s and other functionality of **Alshayun** of which **content author**s and **reader**s can make use. While I speak of these roles in the singular, many individuals may inhabit any given role, and some individuals may inhabit multiple roles.

1.2 Inspiration

After taking a numerical methods course as part of my Computer Science undergraduate degree, I became fascinated with Bézier curves. Whilst researching Bézier curves online,

I came across an article titled *A Primer on Bézier Curves* [12] authored by someone named Pomax. The highly detailed and enlightening article was filled with interactive **applet**s designed to strengthen the **reader**'s understanding of each progressively difficult concept.

Intrigued by Pomax's work, I was inspired to develop a sort of content delivery mechanism designed to allow **content author**s like Pomax author similar articles and deliver them to interested **reader**s. Supporting the development and use of interesting, interactive **applet**s was a priority. I recognized that any such modern application needed to be mobile-friendly, and so I decided to develop exclusively with mobility in mind.

1.3 About the Name

Alshayun was originally meant to be tailored towards mathematics education. However, during the process of developing the application, I realized that I had created a much more generalizable platform capable of servicing any subject and perhaps more. Albeit, I decided to keep the name I gave the application from the time of its inception—a name that carries with it certain mathematical connotations.

Algebra (from the Arabic "al-jabr") is one of the most powerful mathematical devices and fields of study, as well as my personal favorite subject of mathematics. The English-speaking world inherited its written algebraic works from Latin, which came from Spanish, which came from Arabic. This line of inheritance spanned many years and involved merchants and trade routes, among other things. In the original Arabic sources, the word "al-shayun," meaning "the unknown thing," was frequently used to describe the unknown in algebraic equations. Therefore, I found "al-shayun" to be an appropriate name filled with mathematical historical meaning.

To learn more about how "al-shayun" came to be the letter "x" used in modern algebra, check out the article *Why X marks the unknown* by Terry Moore [9].

Chapter 2

Alshayun

2.1 User Interface

When a user loads **Alshayun** for the first time, the application is configured to point at a bogus articles server. Therefore, the application displays an appropriate error to the user (see Figure 2.1). The user may navigate to the settings page by tapping on the blue settings cog in the upper-right-hand side of the screen. Once in settings, the user is able to configure the articles URL to point at a real server, perhaps powered by the **QDS** (see Figure 2.2). Then, upon returning to the home screen, the user may pull down from the top of the screen to trigger a reload of all articles (see Figure 2.3).

After the articles have been successfully loaded, the user is faced with a list of articles where each article is identified by a title, zero or more tags, and an excerpt of the article's content (see Figure 2.4). Furthermore, the user may search for articles using the search box at the top of the screen.

By tapping on an article in the list, the user is redirected to a dedicated page to read the chosen article. Articles are displayed in rich-text with the possibility of embedded images and **applet**s. For example, the Tower of Hanoi article in Figure 2.5 has a series of embedded interactive applets. An example of one of these applets in full screen can be seen in Figure 2.6.

After returning to the articles listing, the visited article is marked as read as indicated by a book icon and a gray text color (see Figure 2.7). The user may swipe an article from

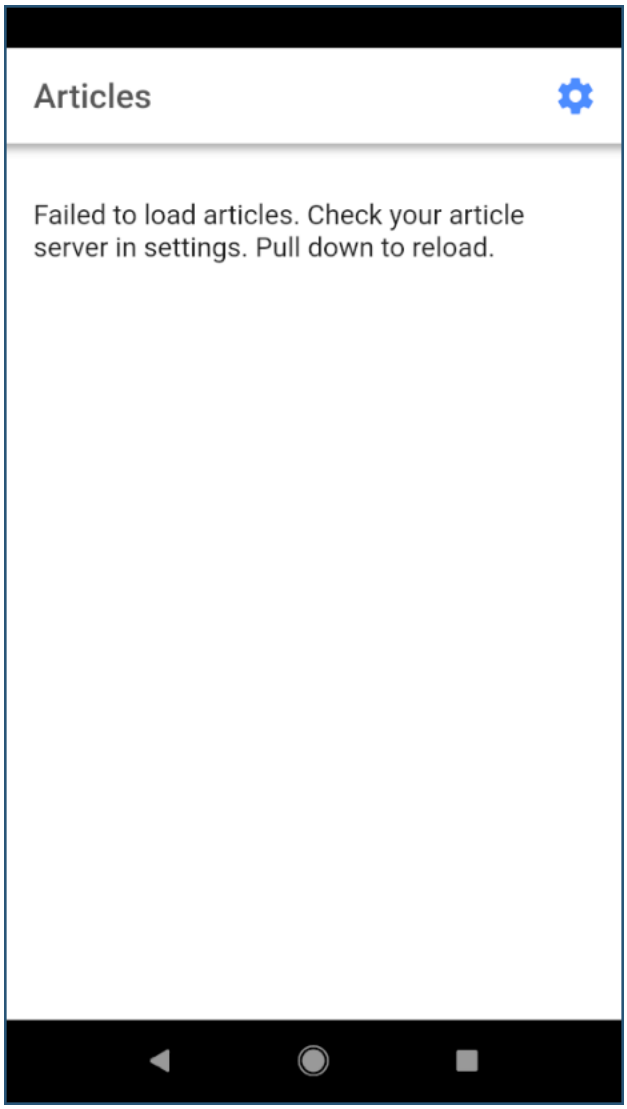

Figure 2.1: Initial startup screen, failed to load articles

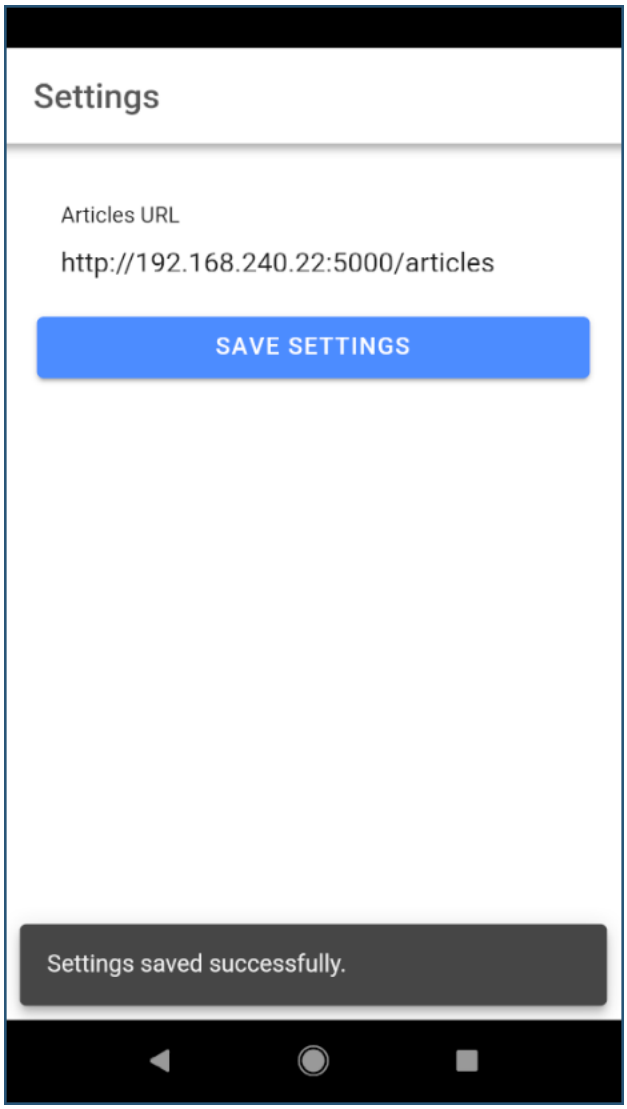

Figure 2.2: Articles URL saved in settings

Figure 2.3: Reloading articles

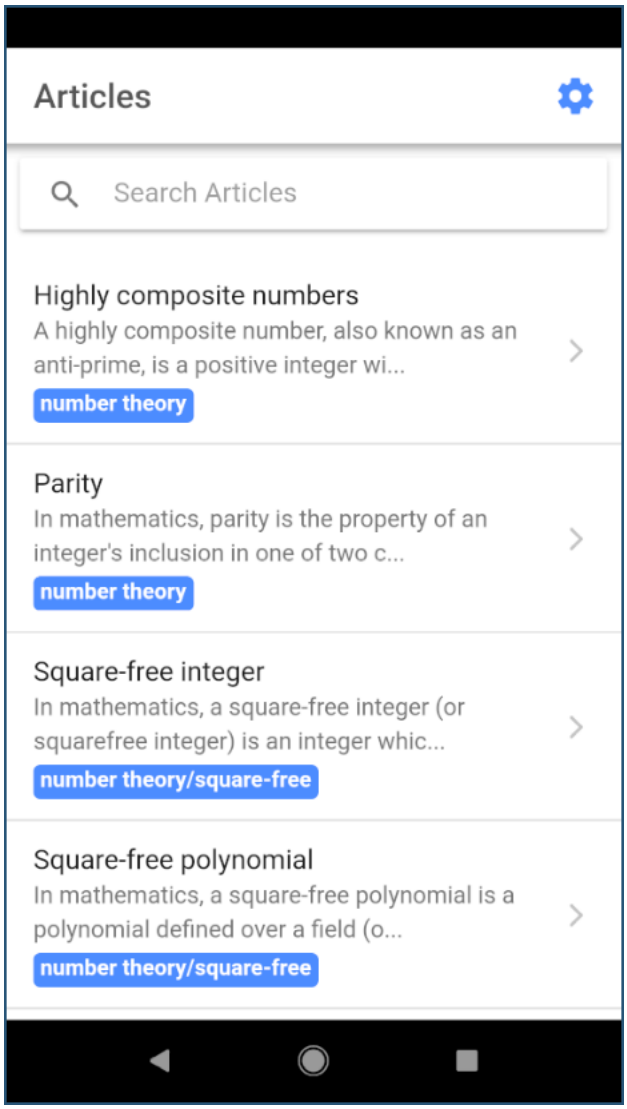

Figure 2.4: List of loaded articles

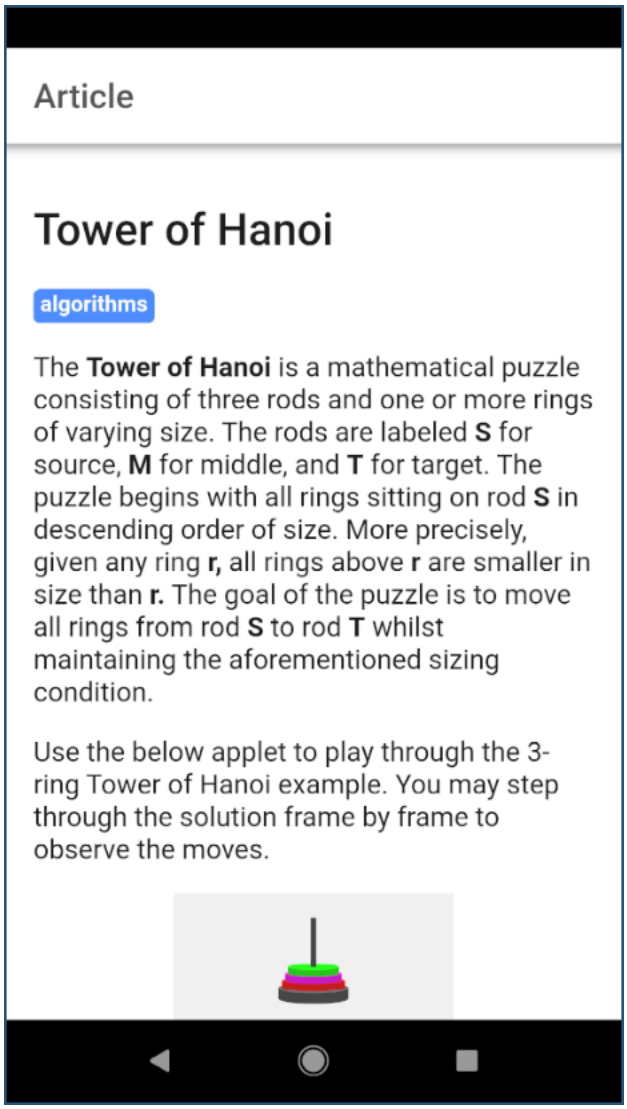

Figure 2.5: Tower of Hanoi article being read

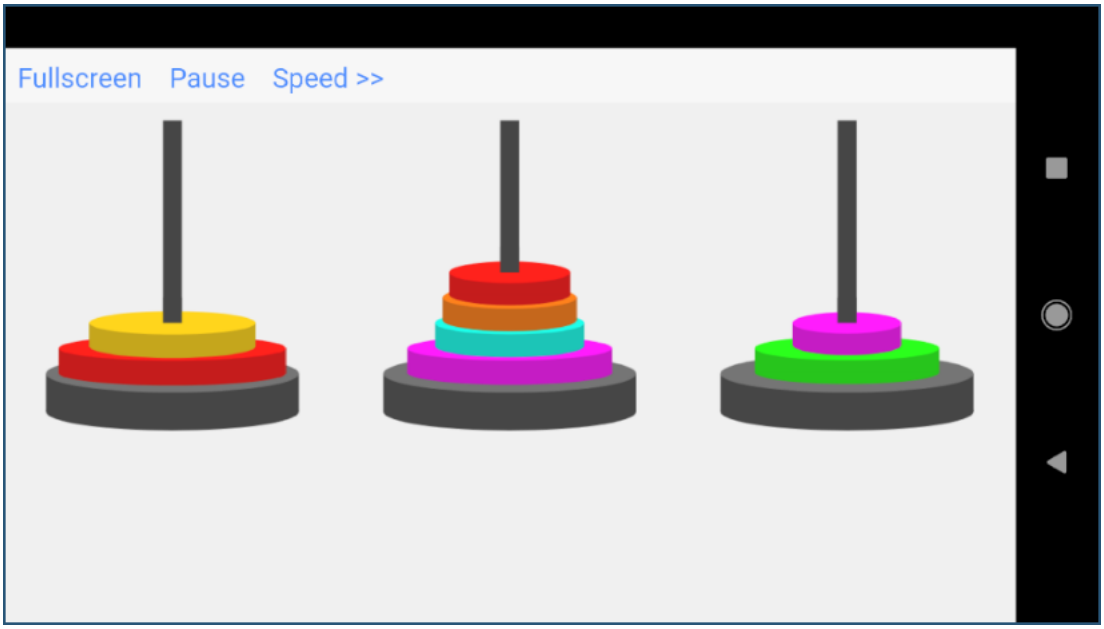

Figure 2.6: Tower of Hanoi applet playing in full-screen mode

the right to access a button allowing the article to be marked unread if desired (see Figure 2.8).

2.2 Caching Strategy

In order to improve application performance as the number of articles grows, the full text of each article is not loaded unless the article has been read. I accomplished this by breaking articles into two pieces: metadata and text. The metadata of an article is its unique ID, title, tags, and excerpt (short description of the article). The text is the body of the article formatted in markdown. When the application loads the list of articles, only the article metadata has been loaded. An implication of this is that article text is not searchable, which is part of the reason I included the excerpt. When a user views an article, the text is loaded on demand and kept in RAM for the duration of the application session.

The manifest (example given below) is stored in **JavaScript Object Notation (JSON),** and consists of an array of objects where each object represents an article. The unique ID is a monotonically-increasing serial number. When a user reads an article, the fact the

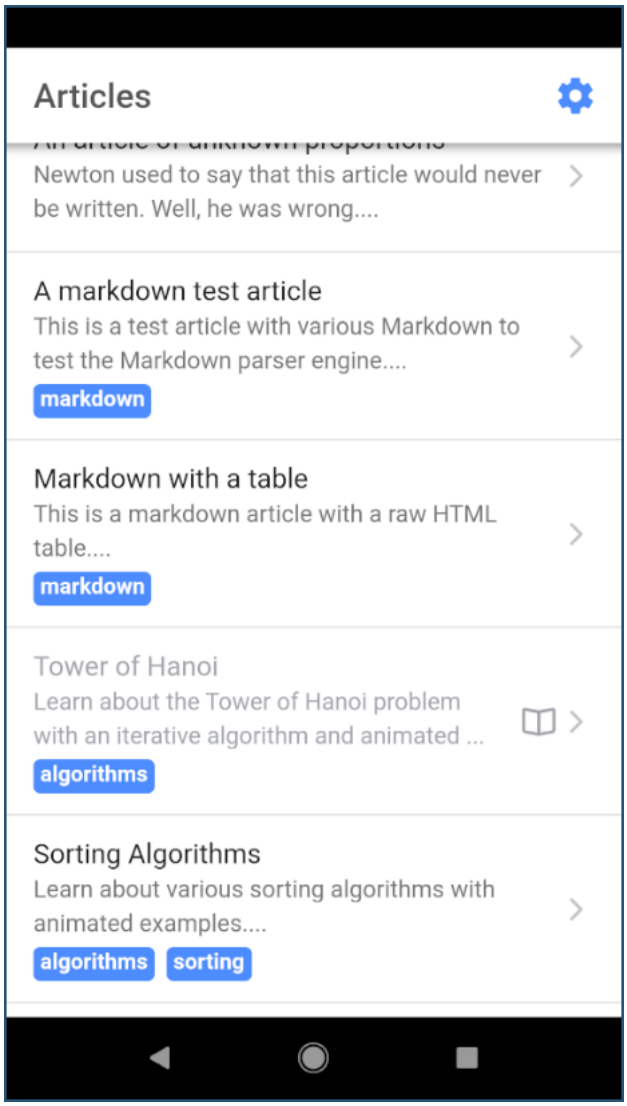

Figure 2.7: Article marked as read in the articles list

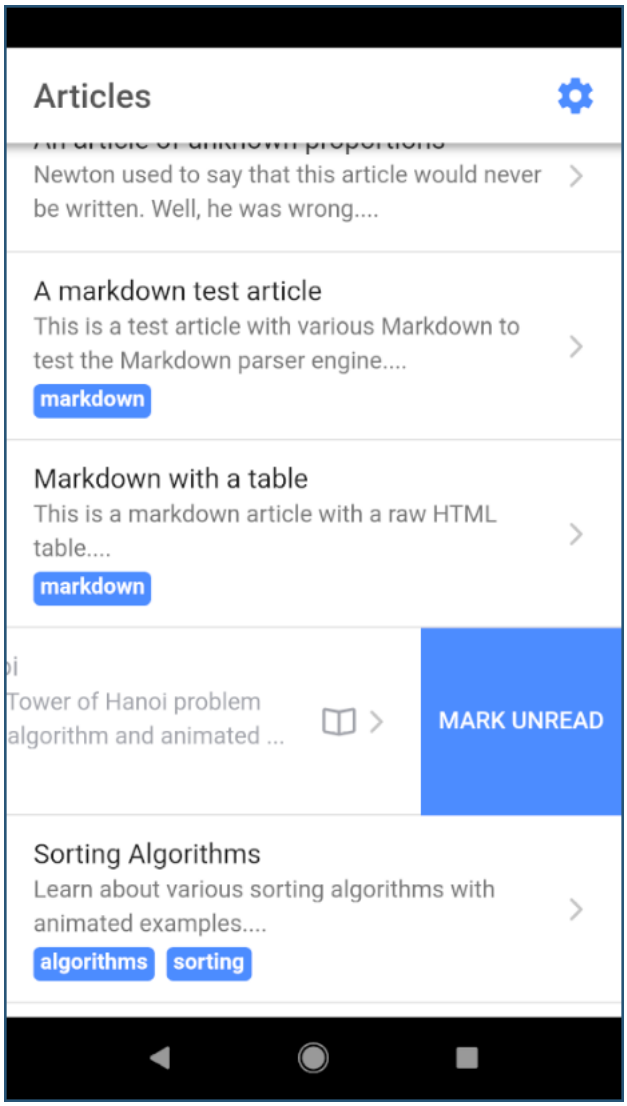

Figure 2.8: Marking an article as unread

article has been read is saved on the user's local device storage based on the article ID. If older (smaller) IDs were recycled, this could cause some articles to be displayed as falsely read.

```
[
    {
        "id": 12,
        "title": "Tower of Hanoi",
        "excerpt": "Learn about the Tower of Hanoi...",
        "tags": [
            "algorithms"
        ]
    }
]
```

2.3 Applets

The **applet**s are the crux of **Alshayun**. **Applet**s are implemented as **Angular** components that extend an **Applet** superclass. The superclass automatically sets up an **HTML** canvas tag with drawing context and implements an animation loop if the subclass requests it. All the subclass needs to do is extend a draw method, which is automatically called by the superclass 30 times per second. The draw method performs whatever drawing needs to be done in the context of the canvas to implement the **applet**'s functionality. Furthermore, the subclass may choose to extend the **applet** toolbar that the superclass sets up. The default toolbar has a button for toggling full screen mode, but more buttons can be added.

Applets are included in articles with a dedicated `<applet>` tag. All **applet**s accept at least two parameters: name and width. The name parameter indicates which specific **applet** should be loaded, and the width parameter sets the width of the **applet** on the screen. If no width is specified, the **applet** assumes a width of 100%. All **applet**s render in the aspect ratio of the device's screen dimensions and orientation. Furthermore, specific **applet**s may implement their own parameters. An example **applet** tag is given below.

```
<applet name="sort"
        width="50%"
        data-method="insertion"
        data-num-bars="50"></applet>
```

Currently, there are two **applet**s implemented in **Alshayun**. Details follow.

2.3.1 Tower of Hanoi Applet

The Tower of Hanoi (see Figure 2.6) **applet** is designed to illustrate the solution to the famous Tower of Hanoi problem. The **applet** supports running between 3 and 8 rings. The user can step through the algorithm frame by frame or have the algorithm run itself with an adjustable speed. The spindles and rings are all drawn mathematically and are two-dimensional despite the three-dimensional appearance.

2.3.2 Sorting Applet

The sorting **applet** (see Figure 2.9) is designed to illustrate the inner workings and relative performance metrics of various sorting algorithms. The **applet** works by sorting between 10 and 100 integers represented by bars of height proportional to the integer. Presently, bubble sort and insertion sort are the two sorting methods implemented. Dark gray bars are in unsorted position, light gray bars are in sorted position, and blue bars have just been targeted for consideration by the algorithm.

2.4 Accepting Contributions

For **Alshayun** to be useful to a large audience, a number of developers need to come on board and contribute code. Particularly, the development of a wide variety of configurable applets is necessary. Because **Alshayun** is hosted on GitHub [1], accepting contributions is easy.

To accept a contribution, a developer must fork the **Alshayun** repository, create a development branch, and work on whatever feature they want. After development, the

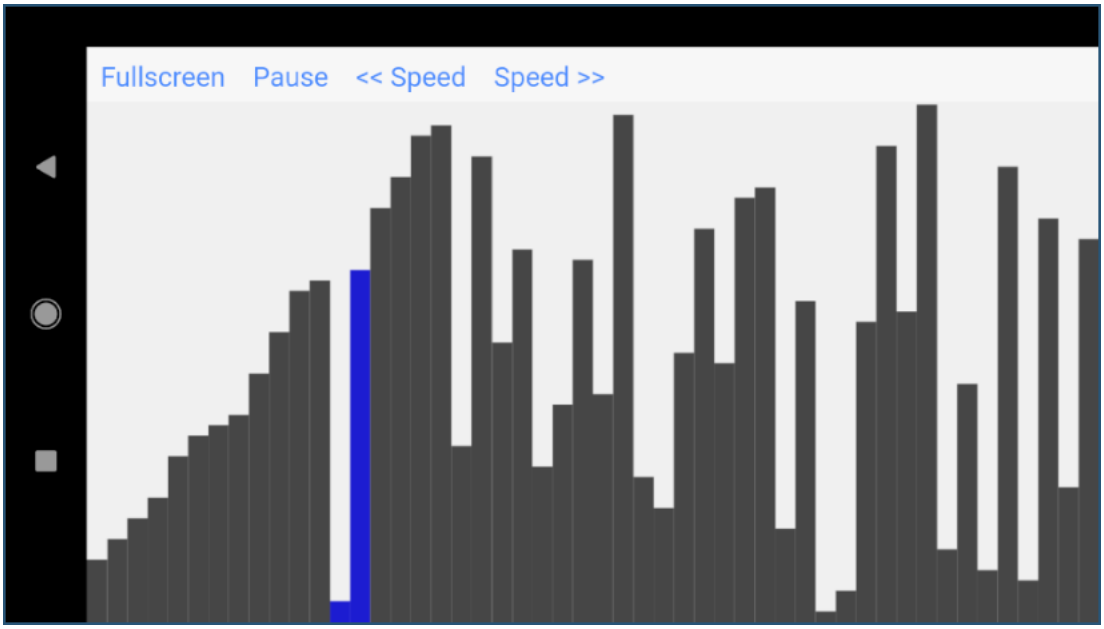

Figure 2.9: Insertion sort applet running on 50 bars

developer submits a pull request to the main **Alshayun** repository, at which point I am triggered to thoroughly review the code, make comments and suggestions, and ultimately accept the pull request. Once the contribution is merged in the master branch, and after sufficient time passes, a release cycle occurs. At the time of a release cycle, a snapshot of the master branch is made and released to the public. Thus all Android users for example have access to the new applets and features developed in between release cycles.

2.5 Building and Running

Building **Alshayun** requires **Ionic**, **Cordova**, and the Android Software Development Kit (SDK) with all of their dependencies. While **Ionic** and **Cordova** support many platforms, I have only tested **Alshayun** with Android, which is why I call it out here. Setting up such an environment is complicated and time consuming. As a result, I have built a **Singularity** container that encapsulates the full development environment necessary to build and run **Alshayun.**

To begin, make sure **Singularity** 3.0 [14] or greater is installed on your computer. I found the container to be useful to all Ionic developers, so I uploaded it to the Sylabs

Cloud Library. To download the image, run the following command from the application root directory.

```
singularity pull ionic.sif library://phphavok/default/ionic:latest
```

Should that command fail for some reason, or if you prefer to build the container yourself, run the following command from the application root directory.

```
sudo singularity build ionic.sif Singularity
```

Once the container has been downloaded or built, run the following commands from the application root directory.

```
mkdir -p sdk/build-tools sdk/platforms
singularity shell \
    -B sdk/build-tools:/usr/local/android/build-tools \
    -B sdk/platforms:/usr/local/android/platforms \
    -p ionic.sif
sdkmanager 'platforms;android-27' 'build-tools;27.0.3'
ionic cordova build android
```

The `singularity` command launches a shell inside the container environment, and you are still inside this container environment upon the completion of the `ionic` command. You may type `exit` or press `CTRL+D` on your keyboard to exit this environment.

The `ionic` command builds an Android Package (APK) file, which you can install on any compatible Android device. If you wish to build for a different platform or version of the Android SDK, you may modify the `sdkmanager` command as desired. You can also have **Ionic** directly install and run the built APK file on your compatible Android device by attaching your Android device to your computer via Universal Serial Bus (USB), enabling USB debugging on the Android device, and running the following command from within the container environment.

```
ionic cordova run android
```

If you don't have an Android device, you may setup and test **Alshayun** on an Android Virtual Device (AVD).

2.5.1 Setting up an Android Virtual Device (AVD)

Exit the container environment and create an additional directory by running the following command from the application root directory.

```
mkdir -p sdk/system-images
```

Then, once again launch a shell into the container environment, and be sure to mount in the additional directory.

```
singularity shell \
    -B sdk/build-tools:/usr/local/android/build-tools \
    -B sdk/platforms:/usr/local/android/platforms \
    -B sdk/system-images:/usr/local/android/system-images \
    -p ionic.sif
```

Select a system image to use for your AVD and install it by running the following command from within the container environment (assuming you chose Android version 27).

```
sdkmanager 'system-images;android-27;google_apis;x86'
```

Once you've installed a system image, run the following command to list available devices and pick one.

```
avdmanager list devices
```

After picking a device (`pixel` in this example), choose a name for your AVD (e.g., `test`), and create it using the selected device and system image installed in the previous steps. By default, AVDs are installed under `$HOME/.android/avd`. If you want a different path, add `-p /path/to/avd` to the `avdmanager` command.

```
avdmanager create avd \
    -n test \
    -k 'system-images;android-27;google_apis;x86' \
    --device pixel
```

Finally, launch the AVD in an emulator.

```
emulator -no-snapshot -avd test
```

Afterwards, you should be able to install and run APK files on the emulator.

2.5.2 Troubleshooting

Building may fail if there is not sufficient space in `/tmp` or under unpredictable networking circumstances. In the former case, allocate more space under `/tmp`. In the latter case, just try the build command again.

Chapter 3

Quick n′ Dirty Server (QDS)

3.1 Motivation

During the early stages of developing **Alshayun**, I included articles in the APK to be installed on devices. While great for initial testing and rapid development, it quickly became clear that such an approach was inflexible and would hinder any future production-readiness of the application. In response, I setup an Nginx Web server on my desktop computer and began storing articles there. However, I wanted the source code of **Alshayun** to be all-inclusive of everything necessary to build, run, and test the application. Thus, the **QDS** was born.

After the **QDS** was built and successfully serving articles to **Alshayun**, I decided to prototype a **frontend** Web interface designed to facilitate the creation and management of articles by **content author**s. Because the **frontend** depended on a RESTful interface provided by the **backend** of the **QDS**, it was only natural to roll the **frontend** into the **QDS** and treat both components as a single deployable unit.

3.2 Backend

The **backend** is written in Flask and serves two functions: deliver articles to **Alshayun** and expose a **REST**ful interface to the **frontend** to allow **content author**s to create and manage articles. The articles are stored in plain-text files on the local disk, and **Flask**

allows easily serving static content through the `send_from_directory` method.

```
@app.route('/articles/<path:filename>')
def get_article(filename):
    return send_from_directory('articles/', filename)
```

In the above code, a request to `/articles/article.1.md` results in the file `articles/article.1.md` relative to the **Flask** application to be read from the disk and served back to the requesting client as the HTTP response. The minimum code to accomplish this task is part of what makes **Flask** so powerful and useful.

As another example, consider the following code which exposes a **REST**ful interface for creating a new article.

```
@app.route('/article', methods = ['POST'])
def create_article():
    if (not request.json) or \
            (not 'title' in request.json) or \
            (not 'excerpt' in request.json) or \
            (not 'tags' in request.json) or \
            (not 'text' in request.json):
                abort(400)
    # Create article object
    article = {}
    article['id'] = checkout_serial()
    article['title'] = str(request.json['title'])
    article['excerpt'] = str(request.json['excerpt'])
    article['tags'] = request.json['tags']
    # Write article file to disk
    f = open('articles/article.' + str(article['id']) + '.md', 'w')
    f.write(str(request.json['text']))
    f.close()
    # Add article object to manifest
```

```
    manifest = read_manifest()
    manifest.append(article)
    write_manifest(manifest)
    # Return status
    ret = {}
    ret['message'] = 'Success'
    ret['id'] = article['id']
    return json.dumps(ret)
```

The `@app.route` decorator prefixing the function indicates that the function should be called if an HTTP POST request is sent to `/article` on the server. The function begins by making sure a minimally valid request has been supplied. The `abort(400)` call is provided by **Flask** and conveniently triggers an HTTP response with error code 400. An article object is created to store information passed in from the request safely, and then the contents of the article are written out to disk using an established naming convention. The `checkout_serial` function keeps track of an auto-incrementing integer called the serial, which is used to name and index articles. The manifest stores the metadata about the article.

3.3 Frontend

The **frontend** is written as a standalone **Angular** application. The **frontend** listens on port 4200 when running, and can be accessed from the URL `http://127.0.0.1:4200/`. Upon accessing, the user is presented with a default sign in screen (see Figure 3.1). After signing in with the default username of `admin` and default password of `password`, the user is presented with a list of articles (see Figure 3.2). If this is the first time the user has signed in, the articles are the sample articles included with the application source code.

The user may delete or view/edit an article by clicking the appropriate buttons next to each article's description. When viewing an article (see Figure 3.3), the user is presented with a field to enter the title of the article, a field to enter a comma-delimited list of tags associated with the article, a field to enter a short excerpt of the article, and a field to enter

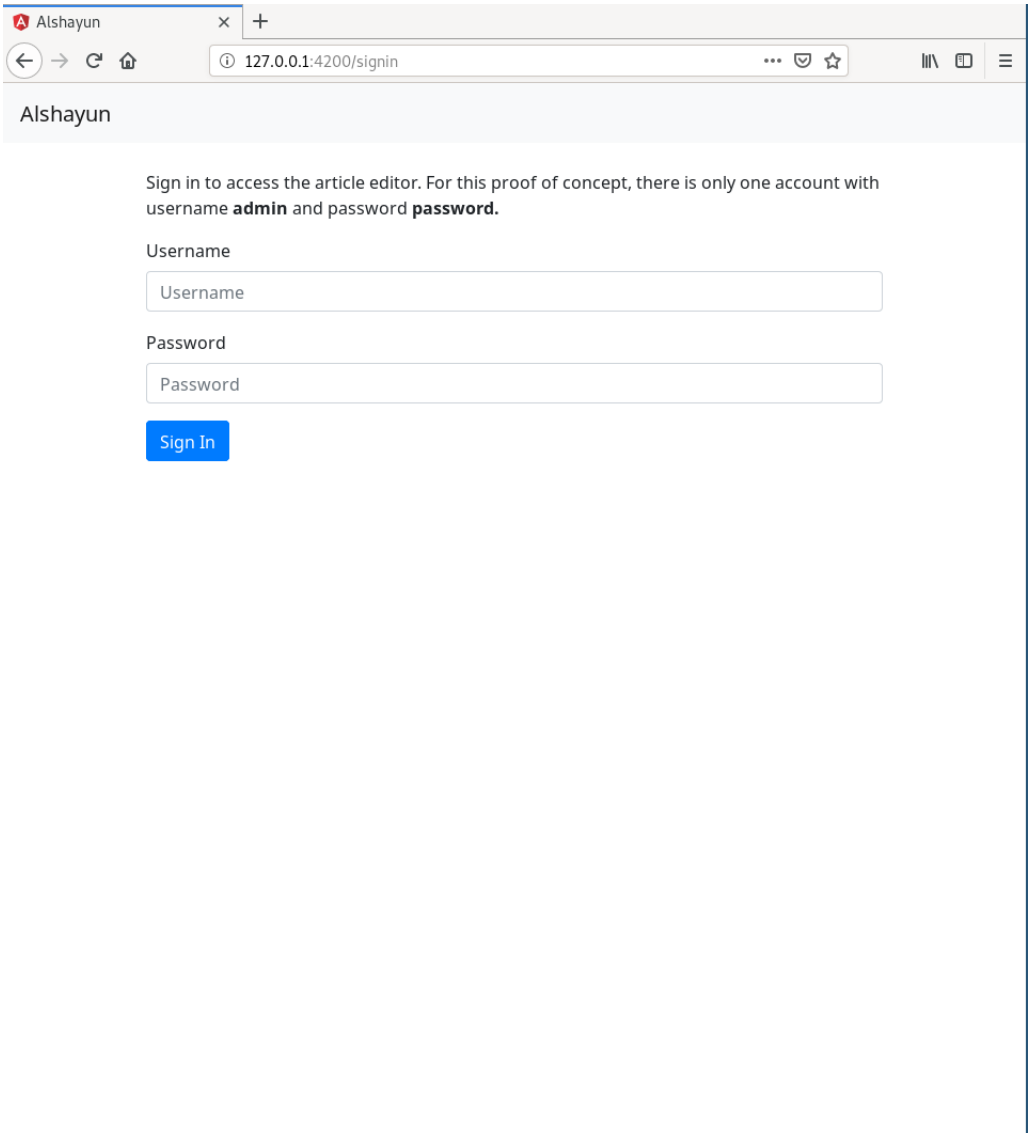

Figure 3.1: Login page of QDS frontend

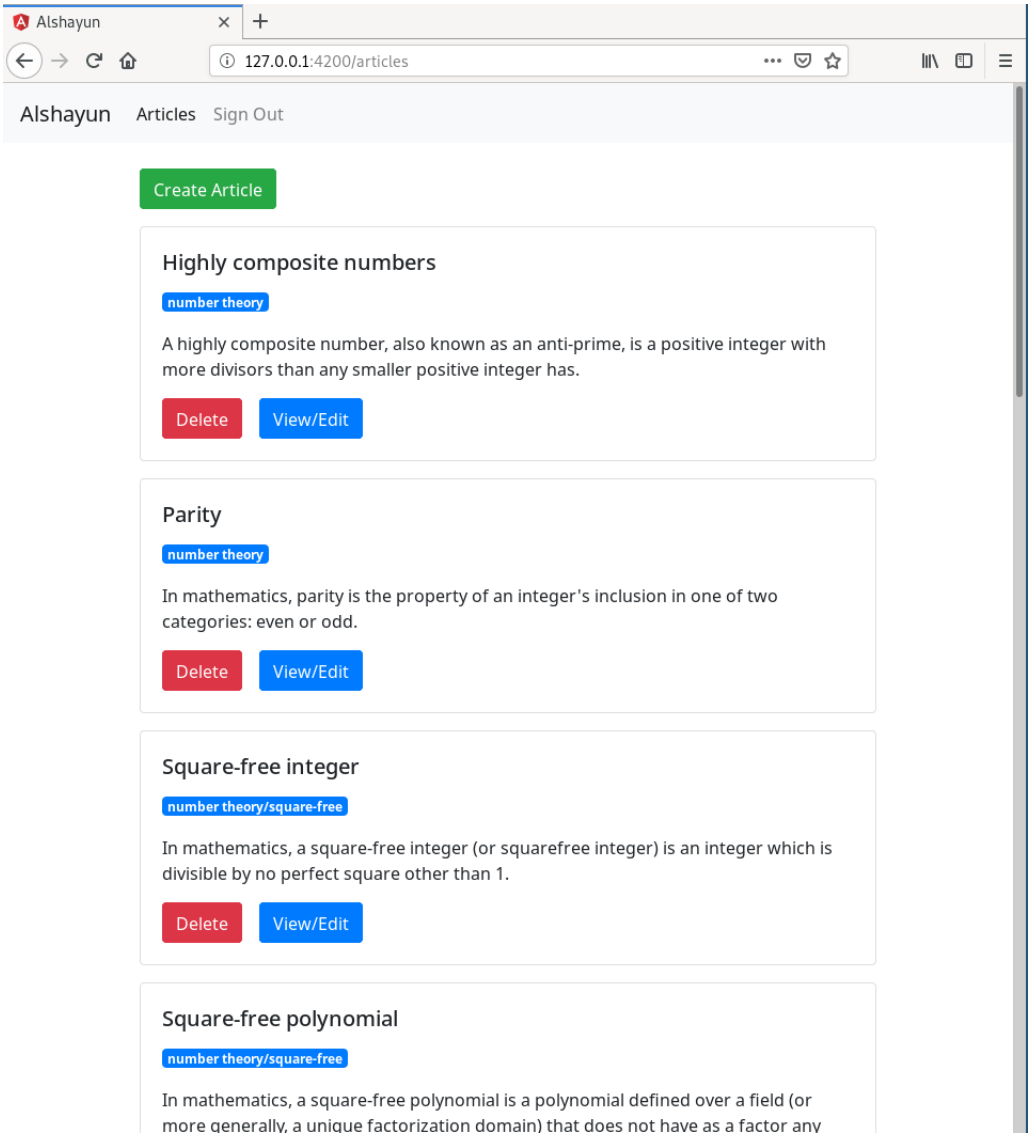

Figure 3.2: List of articles in the QDS frontend

the main text of the article. Articles are written in Markdown [8], and a live preview of the rendered article is presented next to the editable article text (see Figures 3.3, 3.4). The user may also create a new article by clicking the appropriate button on the home page.

3.4 Production Considerations

The **Flask** instance of the **backend** runs a built-in development Web server, as does the **Angular** instance of the **frontend**. However these servers are not designed to handle production scenarios. Therefore, if an **Alshayun** user wants to take the **QDS** into production, some things should be taken into consideration first.

For the **backend**, a real production Web server such as Nginx or Apache should be setup. The Web server can run a Web Server Gateway Interface (WSGI) module which redirects applicable requests to the **Flask** code. Furthermore, the production Web server can and should have signed Secure Sockets Layer (SSL) certificates setup to encrypt requests. Lastly, the current **backend** has no authentication mechanism for the **REST**ful **API**s exposed to the **frontend**. Therefore, anyone can technically create, edit, and delete articles. An authentication mechanism would need to be developed. Options for this include HTTP Basic Authentication, OAuth, client certificate authentication handled by the production Web server, or a variety of other methods.

For the **frontend**, a real production Web server would also need to be setup. Perhaps the same Web server powering the **backend** could be used, which may help aid in security and latency concerns. SSL certificates and an administrative account that isn't `admin/password` is of course necessary.

3.5 Building and Running

The **Quick n' Dirty Server (QDS)** consists of two components: a **backend** and a **frontend**. The **backend** is written in **Flask**, whereas the **frontend** is written in **Angular**. In order to facilitate easy building and running of the **QDS**, I have built two **Singularity** containers: one for the **frontend** and one for the **backend.** Though the **QDS** is techni-

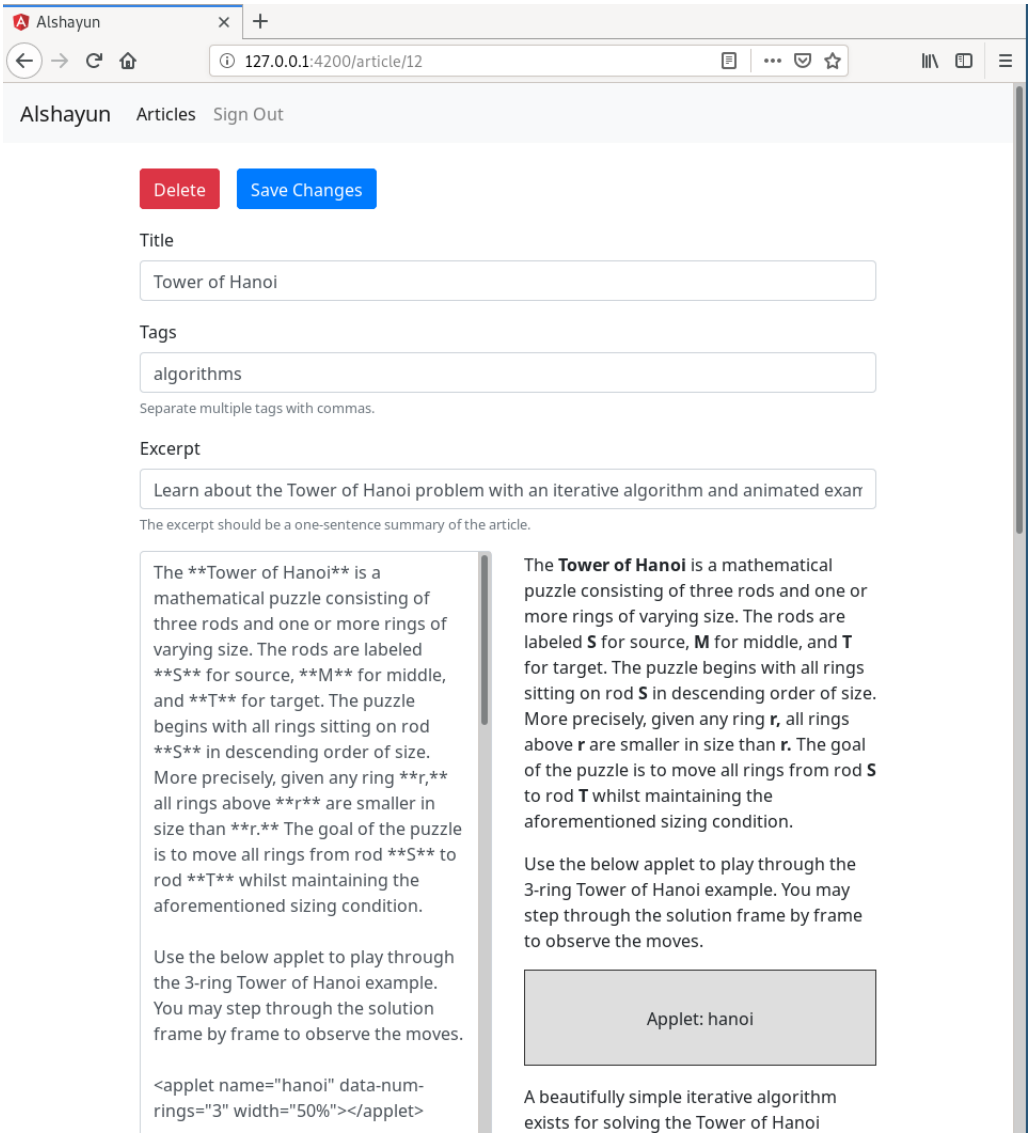

Figure 3.3: Viewing and editing an article in the QDS frontend

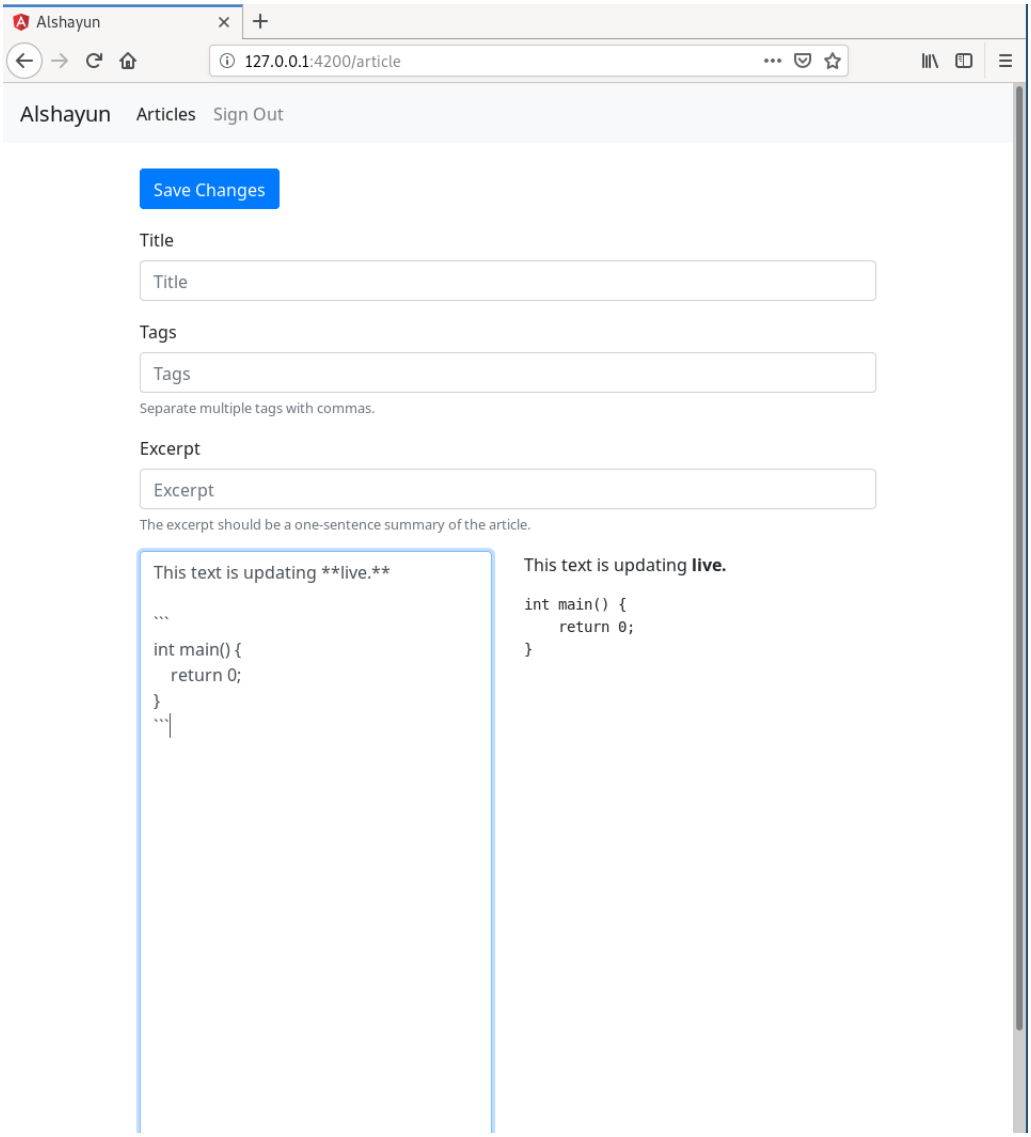

Figure 3.4: Creating a new article in the QDS frontend

cally two components, I refer to it in the singular, and a single Makefile is provided for convenience.

To begin, make sure **Singularity** 3.0 [14] or greater is installed on your computer. Then, run the following command from the application root directory. The command prompts you to escalate to root privileges if you are not already root.

```
make -C qds
```

Upon completion, the **QDS** is built and ready for running. To start the **QDS** as a background service, run the following command from the application root directory.

```
make -C qds start
```

While the **QDS** is running, ports 4200 and 5000 are bound on your computer. The **frontend** can be accessed from the URL `http://127.0.0.1:4200/`. The **backend** runs on port 5000 and is used by the **frontend** and by **Alshayun.**

At any time, you may stop the **QDS** by running the following command.

```
make -C qds stop
```

3.5.1 Troubleshooting

Building may fail if there is not sufficient space in `/tmp` or under unpredictable networking circumstances. In the former case, allocate more space under `/tmp`. In the latter case, just try the build command again.

Chapter 4

Technologies Used

4.1 Architecture

Figure 4.1 illustrates the high-level architecture of **Alshayun**. A discussion of each of the components of the architecture follows.

4.2 Node.js

Node.js (Node) [10] is a **JavaScript** runtime designed mainly to facilitate writing scalable, server-side software in **JavaScript**. Before **Node**, first released in May of 2009, the idea of writing server-side software in **JavaScript** was relatively unknown. Since its inception, **Node** has received much attention and both praise and criticism from **developer**s.

Less than a year after **Node**'s inception, a package manager was released in January of 2010 named the **Node Package Manager (NPM)** [11]. **NPM** has since played a vital role in the development of most **JavaScript** software and frameworks and has transcended its original intent to become a general JavaScript package manager.

While developing **Alshayun**, I never interacted directly with **Node.** However, **NPM** was a vital part of my development process. It is for that reason that I mention **NPM** and **Node** as a used technology.

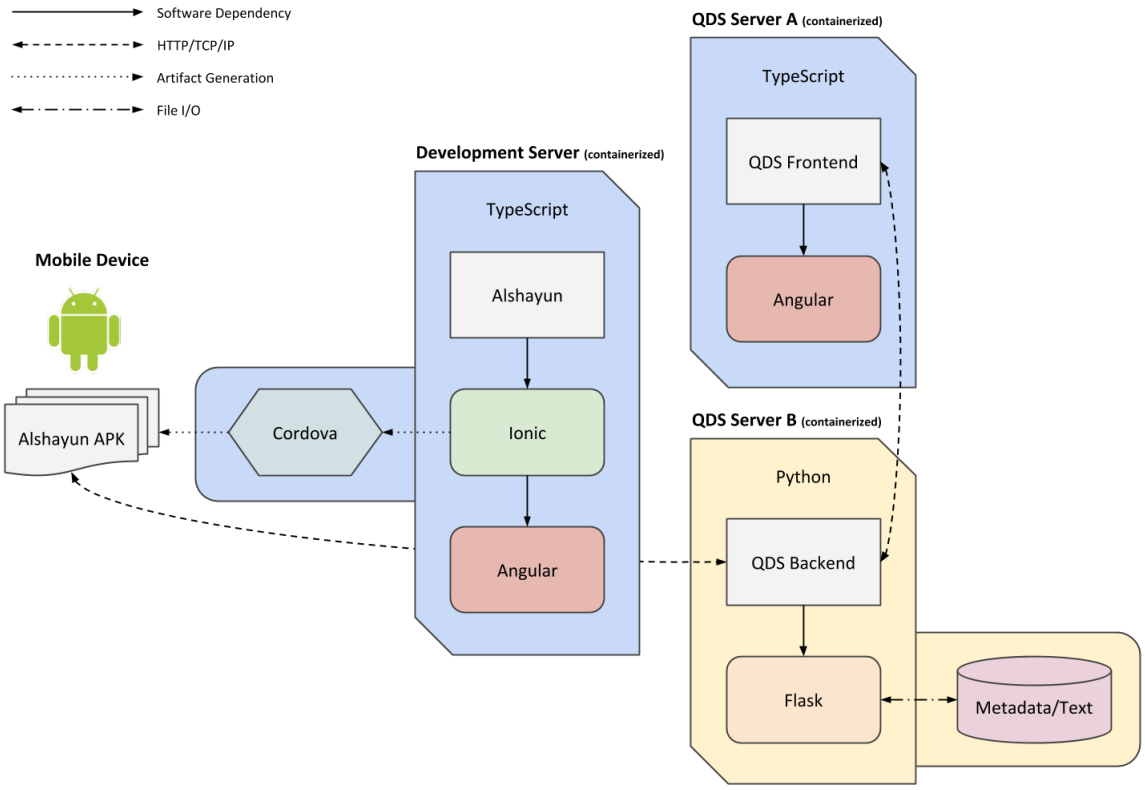

Figure 4.1: High-level architecture of **Alshayun**

4.3 TypeScript

TypeScript [15] is a superset of **JavaScript** that ultimately compiles into **JavaScript**. **TypeScript** complements **JavaScript** with type safety and true object-oriented programming constructs such as classes, access modifiers, and inheritance. In **TypeScript**, every variable and symbol in general have a type, either stated explicitly by the programmer or inferred by the **TypeScript** compiler from usage. The **TypeScript** compiler is available for installation as an **NPM** package.

As the programming language of **Alshayun**, **TypeScript** has been extraordinarily useful in development. In particular, the access to object-oriented programming constructs has allowed me to write clean, modular, and purpose-driven code.

4.4 Angular

Angular [2] is a Web application framework developed by Google and written in **TypeScript**. Relatively new, **Angular** was released in September of 2016, although it's based on a rewrite of its older predecessor, AngularJS. **Angular** is useful for developing responsive, single-page Web applications backed by a full suite of development tools to assist in application development, testing, and deployment. **Angular** is available for installation as an **NPM** package.

The single-page architecture of **Angular** is accomplished by the backing Web-server rewriting URLs and redirecting requests to the root index.html file of the **Angular** application. Angular then uses an internal router module to load the appropriate page of the application. The application can be fully loaded up front, or lazily loaded on demand. Lazy loading is recommended for larger applications to avoid a lengthy initial page load.

One of **Angular**'s strongest development points is its use of the **Model-View-Controller (MVC)** design pattern. **Angular** provides a template syntax for easily binding the model (data, variables) to the view (**HTML** elements such as forms). Thus, if a variable `title` exists and the application has bound it to the heading tag of a page with the syntax `<h1>{{title}}</h1>`, then simply updating the `title` variable automatically results

in the heading of the page being updated. The **developer** need not worry about the details or be concerned with updating the view his or herself. Furthermore, interactive actions such as button clicks can trigger the calling of methods that further update the model.

4.5 Ionic

Ionic [6] is a framework for building cross-platform applications in **Hypertext Markup Language (HTML)**, **Cascading Style Sheets (CSS)**, and **JavaScript**. **Ionic** supports developing an application with a single codebase that deploys to Web, Android, and iOS devices. As of version 4, the **Ionic** codebase is divorced of an underlying **JavaScript** framework, although **Angular** is still the most supported and widely used base for **Ionic**.

Ionic provides a collection of **HTML** tags and **CSS** that generate components similar to **Bootstrap** [3]. Examples of components include buttons, cards, toasts, modals, and lists. **Ionic** components are designed to mimic the look and feel of a native Android or iOS application, complete with built-in gestures and animations.

4.6 Cordova

Cordova [4] is an Apache project that provides a uniform interface for generating device-dependent code for Android and iOS. For example, **Cordova** provides a **JavaScript** interface for interacting with the built-in camera of mobile devices. Thus, the **developer** can write one piece of code for taking pictures and gathering image data, and **Cordova** automatically translates that code into the appropriate programming language and format for desired devices (Java for Android, Swift for iOS). **Cordova** is a vital part of **Ionic**, and **Ionic** provides a direct command-line interface to **Cordova** for building mobile packages.

4.7 Flask

Flask [5] is a **Python** framework for the rapid development of **Representational State Transfer (REST) Application Programming Interface (API)**s. **Flask** allows **developer**s

to prefix **Python** functions with decorators indicating the URL endpoint that triggers the function, the acceptable HTTP methods, and more. **Flask** also provides a collection of helpful methods for generating HTTP responses, handling exceptions, and easily processing HTTP request data. A built-in development server is provided for prototyping purposes.

4.8 Singularity

Singularity [13] is a containerization software that allows users to develop, package, and relocate full-fledged compute environments consisting of an operating system and software binaries and libraries. **Singularity** is one of several containerization platforms in existence. However, it stands apart by being the most secure and, in my opinion, simplest to use. I have a personal connection to **Singularity**, having contributed code to the open-source project and having developed many containers as part of my employment at the University of Kentucky.

Chapter 5

Use Cases

5.1 Classroom Auxiliary Content

One use case for **Alshayun** is as an auxiliary content platform for K–12 schools and universities. Each independent classroom can run its own articles server (perhaps through **QDS** to get started) and notify students to configure **Alshayun** to point to the articles server. Then, the teacher could function as the sole **content author**, providing lecture notes, practice problems, and so on. While **Alshayun** is not designed to protect articles, teachers could implement minor security by only allowing the articles server to be accessed while on the school or university network. The tagging feature could be used to organize subjects.

5.2 Starter Mobile Blog

Another use case for **Alshayun** is as a starter blog. If an upcoming blogger wants to quickly distribute content to readers but hasn't had time to setup a real blogging Web site or build a real blogging application, **Alshayun** could be used as a temporary blog distribution channel.

Chapter 6

Future Work

There are many areas where **Alshayun** can be improved. While not an exhaustive list by any stretch of the imagination, here are a few areas I'd like to improve upon.

6.1 User Accounts

I would like to support user accounts so that **reader**s can potentially save or bookmark articles, comment on articles, and synchronize their personalized settings between devices. This feature would require an overhaul of the **QDS** or preferably a production Web server, and it carries with it certain security and privacy concerns. Furthermore, accounts should be able to have **content author** privileges granted so that users of the **QDS** can sign in independently and author their own articles with a proper set of capabilities and roles implemented.

6.2 Cloud Hosting

Another feature I would like to have is a central production **Alshayun** server hosted somewhere on the cloud (perhaps Amazon Web Services). This cloud server should be used as the default so that users first opening **Alshayun** retrieve articles from the primary central server. This could turn into a large project fast, as the cloud server could implement users and privileges, and would likely lead to a variety of other necessary features.

6.3 Power Efficiency

Finally, I have noticed that **Alshayun** is quite power hungry when many **applet**s are loaded. I would like to invest considerable time optimizing the **applet**s to consume less power and also implement better caching and unloading of unused elements on each page.

Chapter 7

Conclusion

Through my work on **Alshayun**, I have learned the essentials of **TypeScript**, **Angular**, and **Ionic**—three technologies that I likely would not have learned otherwise. I was also able to build interesting **Singularity** containers and contribute back to the **Singularity** community. My work with **Flask** was minor, and mainly drew upon some previous experience. While **Ionic** was the top-level primary framework of my development stack, I learned the most about **Angular**, and my work with **Angular** is likely to impact my future work the most.

Angular is a beautiful **JavaScript** framework, and I never thought I would be the one to compliment or praise a **JavaScript** framework. However, by taking advantage of the programming constructs made available by **TypeScript** and utilizing modern programming methodologies, **Angular** feels like a legitimate software engineering platform. From re-usable modules and components to features like automatic dependency injection, lazy loading and flexible routing, **Angular** is a truly well-rounded development platform.

The most powerful and enlightening feature I learned to use is observables. Observables allow providers to emit events over time while zero or more subscribers can subscribe and unsubscribe from events at any point. For example, the **QDS** has a list of articles with their titles, tags, and excerpts printed. When a user updates an article, I want that list to update immediately as well. In the past, I would have devoted an extraordinary amount of time and code to cleanly realize such a feature. However, with **Angular** I simply developed an observable around the article metadata and used an asynchronous

data pipe to bind the observable to the view. The **Angular** framework takes care of the rest. This is just one of many memorable **Angular** features that I have found useful and enlightening.

Since completing **Alshayun**, I have begun building an **Angular**-powered Web site for my mom's Real Estate business [7]. With the help of **Bootstrap** for styling, the Web site is quickly becoming the most beautiful REALTOR® Web site in the region (others have told me this, I am not just being grandiose). This Web site as it exists would not have been possible without my knowledge of **Angular**, and **Angular**'s full build environment with deployment tools is incredibly useful. At this rate, I suspect many more projects will come.

Bibliography

[1] Alshayun on GitHub. Retrieved from https://github.com/phpHavok/alshayun.

[2] Angular. Retrieved from https://angular.io/.

[3] Bootstrap. Retrieved from https://getbootstrap.com/.

[4] Cordova. Retrieved from https://cordova.apache.org/.

[5] Flask. Retrieved from http://flask.pocoo.org/.

[6] Ionic. Retrieved from https://ionicframework.com/.

[7] Kristi Nickells REALTOR®. Retrieved from https://kristinickells.com/.

[8] Markdown. Retrieved from https://daringfireball.net/projects/markdown/.

[9] Moore, Terry. *Why X marks the unknown.* Retrieved from https://cosmosmagazine.com/mathematics/why-x-marks-unknown-0.

[10] Node.js. Retrieved from https://nodejs.org/en/.

[11] Node Package Manager. Retrieved from https://www.npmjs.com/.

[12] Pomax. *A Primer on Bézier Curves.* Retrieved from https://pomax.github.io/bezierinfo/.

[13] Singularity. Retrieved from https://www.sylabs.io/singularity/.

[14] Singularity 3.0 Installation Guide. Retrieved from https://www.sylabs.io/guides/3.0/user-guide/installation.html.

[15] TypeScript. Retrieved from https://www.typescriptlang.org/.