NEVERNOTE

Author:
Y3507677

Module:
Service-Oriented Architectures (SOAR)

February 11, 2019

Abstract

This paper covers the development and implementation of an online note taking application that has been developed using the Spring framework. The application contains a ReactJS client for taking rich text notes and sharing them with other users of the system. All images in this report are `PNG` files and there should be minimal degradation in quality when zooming.

Note: an Eclipse run configuration has been provided and the client is accessible from `http://localhost:8080`. If for some reason it does not work, then a new maven run configuration is required that is executed from the root of the project and runs the goal `generate-resources spring-boot:run` and this will run the client and server applications.

CONTENTS

I System Design 1
- I-A Architecture . . . 1
- I-B Spring Framework . . . 1
- I-C REST vs SOAP . . . 1
- I-D Methodology . . . 1
- I-E Database . . . 2
- I-F UML . . . 3

II Implementation 5
- II-A Gaining Context . . . 5
- II-B Check Privilege Service . . . 5
- II-C Service Layer . . . 5
- II-D Roles . . . 5
- II-E Validation . . . 5
- II-F Notebook validation . . . 6
- II-G Cross-Origin Resource Sharing (CORS) . . . 6
- II-H Data Access Layer . . . 7
- II-I Controllers . . . 7
- II-J Sharing Notebooks . . . 7

III Authentication 8
- III-A User Registration . . . 8
- III-B Authentication Scheme Considerations . . . 8
- III-C JSON Web Tokens (JWTs) . . . 8
- III-D Security Configuration . . . 9

IV Client Implementation 10
- IV-A Integration with Maven . . . 10
- IV-B Design . . . 10
- IV-C Implementation . . . 11

V Notifications 12
- V-A Server . . . 12
- V-B Client . . . 12
- V-C Improving the system . . . 13

References 14

I. System Design

A. Architecture

The Nevernote application follows a multitier architecture that is commonly found in web applications; summarised in figure 1. This follows the design methodology of a layer only knowing about what is either side of it and results in a cleaner overall architecture. This architecture is comprised of the following:

- **Presentation Layer:** the layer that the user will interaction with the application through. This layer is formed of a JavaScript application built using React [1] for the user interface, Redux [2] for managing the application's state, Babel [3] for compiling the application, and webpack [4] for bundling the JavaScript application in to a deployable format for the embedded Tomcat server that Spring will use.
- **Business Layer:** features Spring Boot [5], for rapid development and ease in configuring the application. React provides an integrated application server (Tomcat) and requires minimal configuration before developing. This layer handles the start of the client's request, passes the request to an authentication filter that Spring Security [6] calls. Authenticated requests are then passed to the controller that the client requested. This controller will then check the validity of the request and pass it to the service layer.
- **Service Layer:** handles interacting with the repositories and controllers. Converting between data types as required (Data Transfer Object to Entity), checking if the client has the authorisation to access the requested resource and performing more complex validation. Bean Validation is performed using the Spring Validation system that is found in the `spring-boot` package. This is based on the Bean Validation 2.0 (JSR 380) specification. This allows for validation during method invocation and reduces the clutter of validation logic throughout the application. Any entity or method argument that is marked with a validating annotation, such as `@NotNull` will be validated during the method invocation. This functionality will be used to implement the logic that checks that a user's requested notebook name is unique.
- **Data Access Layer:** uses Spring Data [7] to access the database and perform the requested operation; translating method calls in to database queries. Spring Data provides the functionality to access databases through the use of Hibernate ORM. This dependency provides a number of functions that can be used for database accessing and data manipulation that is explained in later sections. This layer is used to simplify the interaction between the database and the service layer with the goal of reducing the overall complexity of the application.
- **Data Storage Layer:** is comprised of a H2 [8] in-memory database using the PostgreSQL dialect.

B. Spring Framework

Spring is an application framework for Java and provides an ecosystem that can be developed upon; most of the functionality of the server is provided by Spring. The web functionality is built on top of an Java EE (Enterprise Edition) platform and a large number of the features of this application have been implemented using this revised platform. The Spring framework lends itself largely to this application due to the modules available: security, testing, WebSocket, model-view-controller (REST), dependency injection, and so on.

C. REST vs SOAP

While SOAP was considered as a communication method and design choice, REST was settled upon. This is largely because the server will be stateless and any required information can be fetched from the database or provided by the user. Using REST in the application also allows for it to scale better in the future in regards to implementing mobile functionality; as all areas of the program will be mobile ready. Using REST also results in less data being transferred, faster performance and for more user-friendly APIs to be opened up to other developers in the future. The advantages that SOAP provides in terms of security and distribution can also be implemented in to a REST application.

D. Methodology

Throughout the design and implementation of this application, several rules were followed to help reduce the overall complexity of the application and to build an application that is easier to extend in the future; such as, if a client was to request some further functionality.

- **Don't trust what the client sends:** even if data has been validated by the client, it should be re-validated again on the server in case it is malicious. This helps to prevent anyone performing a request not using the client and attempting to perform a malicious operation.
- **Always validate the token before proceeding with a request:** on every request validate the request token before invoking the target method. Doing this at the earliest possible stage can prevent damage.

- **Be careful when using queries that contain user data**: a user could be attempting SQL injection and this must be guarded against. Using `query.setParameter` the JDBC driver will escape the malicious code [9].
- **Update the token once the request has been approved**: this will help to reduce the complexity of the application.
- **Check that the user has permissions to access the requested resource:** for example, check that the user is allowed to access the requested notebook ID.

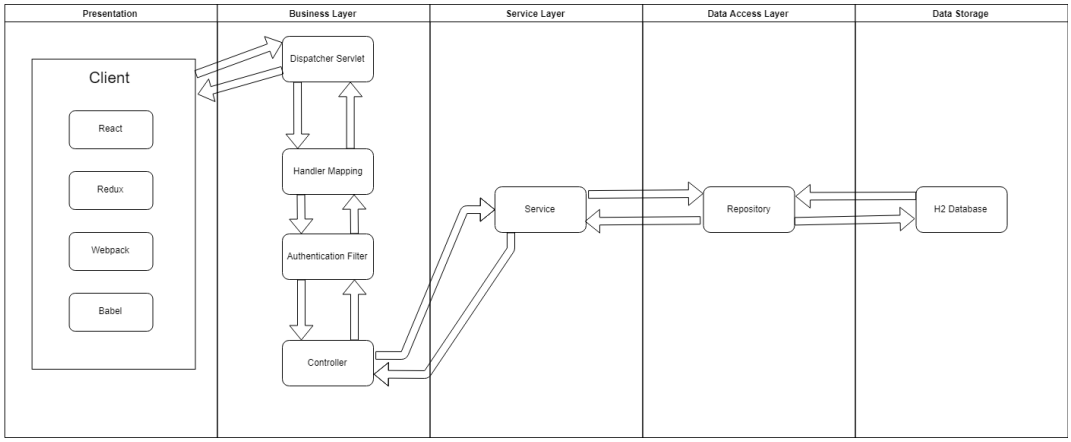

Fig. 1: Application architecture

E. Database

From the specification provided, several entity relationships can be deduced in order to design a database schema. As follows:

- **User**: alongside the user details, a user has many notebooks that belong to them. In order to use the application an account must be created and the user's password must be hashed before persisting.
- **Notebook**: a notebook contains many notes. From this, the notebook can deduce what notes it contains.
- **Shared notebooks**: a notebook can be shared with another user. Thus, the target notebook is required and the user that it has been shared with. From the target notebook, one can find out the creator of that notebook so they can find out what notebooks are shared with who.
- **Roles**: for Spring Security to work correctly user roles are required. Implementing this from the start also allows for the program to become more scalable through requiring a user to have a certain permission set in order to execute some functionality. Available roles will be `ROLE_USER` and `ROLE_ADMIN` and will be automatically imported during application initialisation through the use of the `data.sql` script in the application resources directory.

This specification forms what entities must be implemented in to the application. An in-memory database (H2) [8] is used for persistence in the application. Alongside this, an Object/Relational Mapping (ORM) framework will be used to translate the Plain Old Java Object (POJOs) entities to database entities; Hibernate ORM is used [10]. The schema in Figure 2 shows the schema that has been generated by Hibernate processing entities that existing within the application. Entities are denoted by class annotation of `@Entity` and the fields within the entity can create relationships with other entities. Entity relationships are specified in entities through the use of Hibernate annotations: `@OneToOne`, `@OneToMany`, `@ManyToOne`, alongside additional properties required to form the correct relationships; see package `com.type2labs.nevernote.jpa.entity` for more information on the implementation.

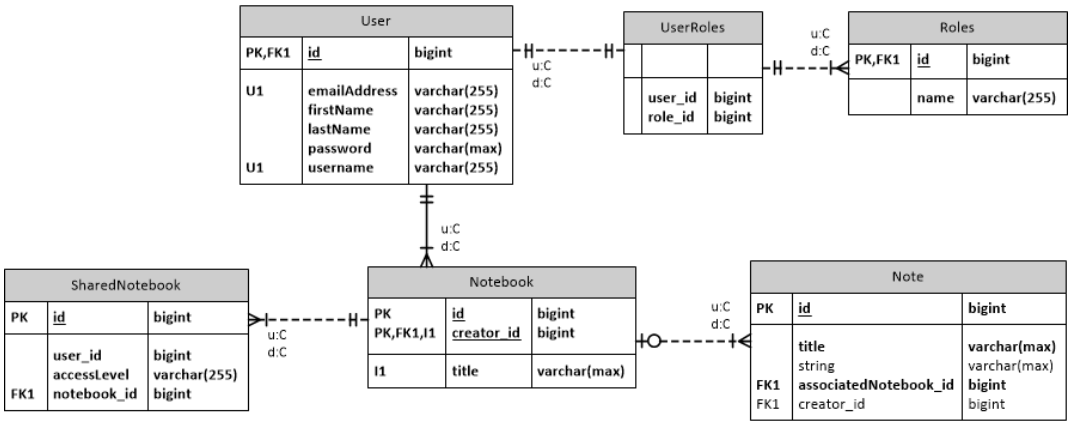

Fig. 2: Hibernate generated schema

Figure 2 details the implemented schema from the entities that were required. Note: the text on the lines details the update (u) and delete (d) cascade type for the parent. As the user will be able to delete their notes and notebooks, the cascade types were considered as an option for performing the delete operations.

An assumption was made that notes that are starred are stored in a non-modifiable notebook named *Starred*; if the user stars a note then it is automatically moved to that notebook. Thus, the database does not treat starred notes any differently, the application itself does.

F. UML

For each entity type that exists in the application and is exposed to the client a separate controller exists that has it's own mapping. As such, four controllers exist in the applicaiton: `AuthenticationController`, `NotebookController`, `NoteController` and `UserController`. In Spring, a controller is a class annotated with `@Controller` that maps a path and HTTP request type to a given method. For example, the `NotebookController` is annotated with `@RequestMapping("/api/notebooks")`, this gives it the path provided and then the methods inside the class have their own annotation that maps the method to another path and HTTP request type. The UML diagram in figure 3 describes the exposed endpoints that are available to the client that have been built up using this application logic.

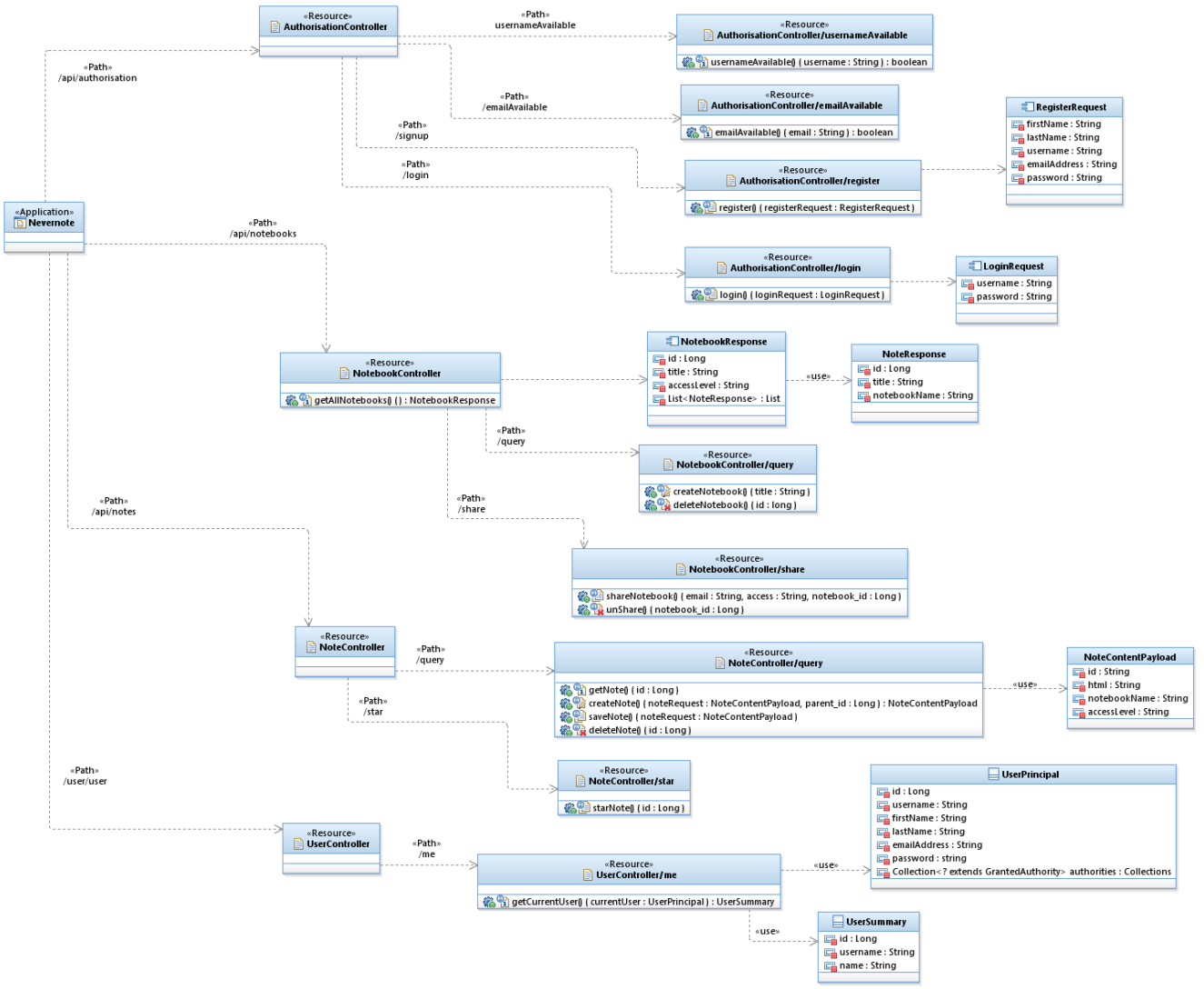

Fig. 3: Controller architecture

To reduce clutter in the diagram, no exceptions are noted in. This is because any method that accepts parameters can throw an `IllegalArgumentException` if the arguments are empty or if an argument is provided but the corresponding entity cannot be found a `ResourceNotFoundException` is thrown. Both exception types map to a HTTP 400 Bad Request response with the provided exception message. If a request parameter is not provided and no other method signature matches the request then a HTTP 404 Not Found is returned as no matching method can be found.

Controllers that require or return specific payloads (`RequestBody` in Spring) have the corresponding class noted in the UML through the «use» notation. Only the data requested is sent back to the client through a Data Transfer Object (DTO) and as such, database entities are not used for transferring data. The payloads that correspond to various entities are in the UML.

The UML is broken down in to a class-method-path format that presents an easy-to-read version of the exposed functionality. For example, the `UserController/me` resource corresponds to the path `api/user/me` and all methods that use this path are displayed in the box itself; this is the same for all the other resources in the UML. Controller methods follow the REST Create Read Update Delete (CRUD) design pattern and the corresponding HTTP request types are implemented; this are denoted by the icons in front of the methods. The `UserPrincipal` is explained in section III.

The service and data access layer are not detailed in the UML. This is due to the number of methods that are in each of the classes, however, they follow a simple design of completing the functionality that was requested by the user. For example, `getAllNotebooks()` that is exposed in the `NotebookController` queries the database for all the notebooks that user has. As the notification service is implemented in the service layer, this too is not detailed in the diagram; this is detailed in section V-A.

II. Implementation

Following the architecture that is detailed in the system design section, the server was implemented layer by layer. Each layer has it's own set of processes to follow to ensure security and data integrity; for example, the controllers validate the inputs that they have received, services check that the requester has the authority to access the data, and the data access layer performs checks on the data that it receives. This resulted in high coupling and low cohesion of the application and allows for the various classes to be used by any other class as long as it provides the correct inputs.

A. Gaining Context

As detailed in section III, the current `UserPrincipal` must be set in order for a request to access a protected resource. From this, one can access the currently authenticated user with the following:

```
(UserPrincipal) SecurityContextHolder.getContext().getAuthentication().getPrincipal();
```

This allows for all areas of the server to gain context in to what it is dealing with, for instance: delivering the correct set of notebooks back to the client when they request to see their notebooks.

B. Check Privilege Service

The `CheckPrivilegeService` checks whether or not the currently authenticated user has access to a given resource. This functionality was implemented early on in to the service layer to stop clients from accessing resources that they do not have access to. In addition to this, having a multi-tiered application allows for the service layer to dictate who has the access to what resources. This class throws a `AccessDeniedException` if the user is not the notebook owner or if the user has not had the notebook shared with them. Having this functionality in it's own service allows for further checks to be implemented in to a single point and for them to be reflected elsewhere in the application.

C. Service Layer

The service layer interfaces with the business layer and the data access layer. Performing additional validation on both requests from the business layer components and the returned data from the repositories. If an entity is not found, the the service layer components will throw a `ResourceNotFoundException`. If this exception is raised, then the existing container transaction is rolled back and this exception automatically maps to a HTTP 404 Not Found response to the client and is handled accordingly with a user-friendly message. Though, it should never happen.

The service layer is perhaps the most important layer in the application as it translates a request from the client in to one that manipulates their data. As such, security has been implemented where possible and all known data handling issues have been resolved accordingly; such as ensuring that notes are deleted before the notebooks are in order to ensure that no database exceptions are raised. Duplicate requests have also been prevented when a user attempts to share the same notebook twice and so on.

Separating the application up in to multiple layers has helped to ensure that edge cases are taken care of in a single point and functionality is not re-implemented. For example, as the username and email address is lowercased, all subsequent checks that require this should come from the `AuthorisationService` as checks will be sure to treat the fields in that fashion when checking for equality.

D. Roles

As mentioned in the system design section, there is a requirement by Spring of having a roles assigned to users in the application. This implemented functionality would allow for an administration console to be implemented in the future and for it's functionality to be easily exposed. To use the roles functionality on a controller, either the class itself the target method has to be annotated with `@HasRole("ROLE_NAME")` and the functionality takes effect.

E. Validation

Spring uses Inversion of Control (IOC) and Dependency Injection (DI) to manage classes (enterprise beans (EJB) in the application; an enterprise bean is a class that encapsulates business logic in the application [11]. During dependency injection resolution the `ApplicationContext` of a Spring application initialises all beans and generates metadata for all of them, metadata is built around the class, including the methods that it contains and metadata surrounding the method; including annotations that configure a method to an endpoint. These classes live in a container and when required they are delivered to the corresponding thread that a request is serving on and the methods inside the class are invoked using reflection. Adding in this DI framework exposes a number of life-cycle events that can be extended to serve additional functionality. The Spring validation framework hooks in to these life-cycle events and before the target method is invoked the method arguments are

checked to see if they contain any validation-related annotations. If so, validation is run on the target object. For example, the `AuthorisationController` uses this functionality to see if the provided `LoginRequest` is valid through a method signature of `@Valid @RequestBody LoginRequest request`. Inside the `LoginRequest` itself are the following:

```
@NotBlank
private String userName;
@NotBlank
private String password;
```

`@NotBlank` is a validation identifying annotation. If validation fails here then the method is not invoked and instead a HTTP 400 Bad Request is returned to the client along with what validation messages were generated; such as "Field userName cannot be blank". However, validation should only fail here if the request was malicious as all validation is performed on the client side too.

F. Notebook validation

Through the use of a custom validator and annotation the unique notebook constraint validation was implemented: `NotebookNameUniqueValidator`. This is a class-wide annotation, so the notebook itself is validated and not a field within it, that is annotated directly on the notebook. The validator performs the following checks: if the title is empty, whether or not the user has any other noteboooks and if the user's other notebooks contain the requested title. From these checks, the validator returns the outcome and if a violation is found then the default message contained within the validation annotation `NotebookNameUnique` is returned: *A notebook's name must be unique*.

This entity validation is performed in the service layer of the application and is manually invoked. This is due to requiring a higher level of control over how Spring handles validation errors. While the validation result can be passed in as an extra argument to a method, this does not lend itself to being a clean solution. Instead, validation is manually invoked and a custom `ValidationException extends RuntimeException` is thrown by the offending method with the argument as the list of validation messages. A global exception handler has been registered with the application and it invokes the corresponding method that is registered with that exception type. As follows:

```
@ExceptionHandler(ValidationException.class)
public ResponseEntity<Object> handleValidationError(HttpServletRequest req, ValidationException ex) {
   return ResponseEntity.status(HttpStatus.BAD_REQUEST)
         .contentType(MediaType.APPLICATION_JSON).body(ex.getMessages());
}
```

This results in a clean violation message being returned to the client; demonstrated in figure 8. As this exception is only thrown in the service layer, no reflection induced validation by the container results in this method being called. Another exception handler for the `BadRequestException` is registered in this class.

G. Cross-Origin Resource Sharing (CORS)

Throughout the development of this application, the client has been running at `http://localhost:9090` and the server at `http://localhost:8080` and it would be in a similar fashion in a production environment; figure 4 explains how CORS works. As a result of this, cross-origin is occurring as the applications are running on different ports. To allow for the client to run correctly, the server has to make allowances in it's design. As such, the security policy for the server has enabled a `CorsFilter` and the `WebMVcConfig` has added mapping for all HTTP methods to be used; by default, only GET, HEAD and POST are enabled by Spring. In addition to this, when the first JWT is returned, an extra HTTP response header must be set in order for the client to be able to access the token: `responseHeaders.set("Access-Control-Expose-Headers", "Authorization")`.

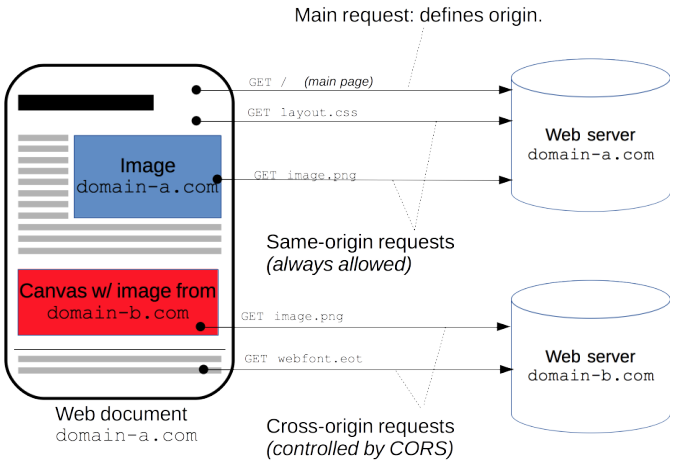

Fig. 4: Cross-Origin Resource Sharing Diagram [12]

H. Data Access Layer

As mentioned earlier, the Spring container largely works through reflection and initialisation-time metadata generation and the repositories are no different. In Spring a repository is an interface that provides access to the database; with two type parameters: entity type, and the primary key type. An example:

```
@Repository
public interface NotebookRepository extends JpaRepository<Notebook, Long> {
   List<Notebook> findAllByCreatorId(Long id);
   Notebook findByTitle(String s);
}
```

This interface provides access to the notebook table and a notebook has a primary key of type `Long`. The method signatures in the interface denote what operation is performed via the method name itself and the arguments; the first method will equate to a query of `SELECT nb FROM NOTEBOOK nb WHERE nb.creator.id = :id`.

While it is possible to implement the queries yourself, the Spring Data repositories are exceedingly powerful and provide a much cleaner method of accessing the database; for more complex queries it is possible to use the underlying API.

The repositories that have been created are used in the service layer only for performing data access and manipulation operations.

I. Controllers

In Spring, a controller maps a method to an exposed end point and it intercepts the request. Performing required operations before handing the data to the service layer and then returning the response. The client application always requires some form of response from the server, be it a message or a status returned. As such, there exists an `ApiResponse` class in the server for this purpose. This returns a message and a status; generally `true`, but the flexibility is there in case there is some kind of "soft error" that can take place in the future. Methods in controllers that only return a status and no payload, return a `ResponseEntity<?>` that bundles the information. This allows for more flexibility in the HTTP response; such as, providing an object creation URI to the client when registering.

Controllers accept and returns the most minimum amount of data possible in order for both the client and server to operate effectively; without requiring resulting in complex database queries. As such, several request and response payload classes exist. The request payload classes contain the required validation setup in order to re-validate what the client has supposedly validated and the methods trigger the validation once more.

All controller methods that accept parameters validate them before continuing with their operation. Following the rule of "*all data from the client is malicious until tested*" methodology to ensure that the server only acts on valid requests. Without the application performing this kind of checks, a user could gain access to all the data in the database and wreak havoc.

J. Sharing Notebooks

When a client requests to share a notebook, several checks are performed: is the user attempting to share another user's notebook (i.e, one that was shared with them), is it the starred notebook, does the recipient's email address exist and is the user attempting to share it with themself. If all of these checks pass, then the user is allowed to share the notebook.

Notebook sharing functions by persisting an entity that contains the recipient's user object, the notebook and an access level. When a user loads their Nevernote client, a request is made that fetches all notebooks, including shared ones. An access level is assigned by the server to all notebooks that are served to the client and so it can then use it to determine whether or not to enable editing in the client. Subsequent server-side checks are in place to determine whether or not the client is allowed to save a notebook.

The user is allowed to unshare notebooks, however, this revokes the notebook from being shared with anyone. Future work here is left to revoke it only from a certain user or to change their permissions. A nice to have here would also to be able to see an edit history of what was changed, when and by who.

III. Authentication

A. User Registration

Since the specification calls for a modern web application, it was decided upon to also implement a full user registration scheme in to the application; including password encoding. When a user is attempting to register with the application, their username and email is checked against the database to check that it is valid, if it is not then a message is returned informing them of this. As the username and email must be unique, the corresponding database columns are set to be uniquely indexed. Both client and server side validation occurs on the entities to ensure that no exceptions are raised throughout this process. If the request passes validation then the registration process can proceed.

The application's security configuration has a bcrypt password encoder registered for encoding passwords. During user registration, the user's email and username are both forced to be lowercase and the password is encrypted using this encoder and then persisted in the database. Subsequent checks against the user's details follow this pattern to ensure continuity. In addition to this, the `User` entity's password field is annotated with `@Lob` to ensure that the database allows for the maximum column length.

Note: in other applications one would treat the password as a `char[]` so it can be nulled once it has been used. Unfortunately, the password's life duration in the `Servlet` cannot be reduced as the `HttpServletRequest` returns the password as a `String` [13].

B. Authentication Scheme Considerations

Authentication in Nevernote is implemented using Spring Security and JSON Web Token [14]. However, before implementing it, several other authentication schemes were considered.

- **Basic**. Sending the username and password in the `Authorization` header in a base64-encoded string. This was considered against as there are no details of the deployment server in the application specification. This can lead itself to a man-in-the-middle attack quite easily if not deployed with a Secure Sockets Layer (SSL) certificate.
- **Cookies**. Using cookies for the authentication has many advantages over basic authentication, however, it does not work well for cross-domain requests and as such will not be used here. The application is designed with Cross Origin Resource Sharing (CORS) in mind to future proof the application and this scheme will not work here.
- **Tokens**: Tokens solve the above problems and are highly flexible and can be persisted. They best fit the scheme here of a secure, single sign-on and provide cross-domain flexibility.

C. JSON Web Tokens (JWTs)

JWT is a compact method of transmitting between two devices and works by base64 encoding three parts: a header, a payload, and a signature; all separated by dots (.). These three parts contain the following:

- **Header**: the algorithm used to sign the token.
- **Payload**: a subject (user ID), the time the token was issued (epoch) and the time it expires (epoch).
- **Signature**: the algorithm, payload and a secret key is used to produce a signature that can be verified. The secret key for Nevernote is: ASuperStr0ngJwtAuth3ntcatonSecretK3y.

When the user successfully authenticates with the system this information is then taken, signed, returned to the user in the response headers with a header of `Authorization` and a value that is prefixed by `Bearer` and the client sends a request header of the same value in future requests. The control flow is detailed in figure 5 when a user is signing up; the last two steps correspond to an initial sign in.

This results in a header of: `Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9 .eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ .SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c` being sent with every request.

The Spring Security filter chain exposes a number of filters, notably the `OncePerRequestFilter` that can be used for authentication request filtering. This is called before the servlet is and can be used to cancel a request if the authentication information is invalid. When the client makes a request to a protected resource, the implementation class `AuthenticationRequestFilter`'s `doFilterInternal` method is invoked and the request headers are checked for a valid token. If one is found, then the current user is retrieved from the database using the user ID that resides in the JWT and then the currently authenticated `UserPrincipal` is set to the current user against the `SecurityContext`; the *current* is in reference to the current thread of execution as each request is given it's own thread. If no JWT is provided then no `UserPrincipal` is set. If it's invalid, an exception is raised. If the token is valid, then it is refreshed so it is always up-to-date. Some applications have an endpoint that is used specificly for refreshing tokens; i.e, the client makes a request and if it is denied then the token is refreshed, or the token is refreshed before the request is made. However, this was decided against for simplicity in this application.

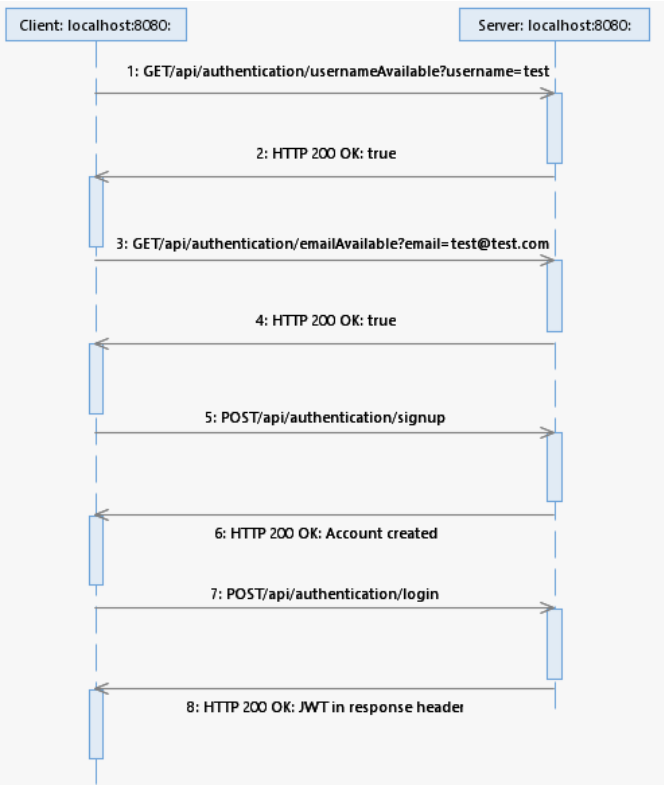

Fig. 5: Sign up control flow

D. Security Configuration

Since an `AuthenticationRequestFilter` is in place, routes need to be exposed in order to be able to use web sockets, access the H2 console and be able to register and login to the application. This is achieved through extending the `WebSecurityConfigurerAdapter` class and overriding the `configure` method; this is also where the password encoder is registered.

Spring Security allows for certain paths to be accessed when there is no registered `UserPrincipal` against the active thread. Using the `configure` function the following rules were put in to place:

- `headers().frameOptions().disable`: allows access to the H2 security console. This is for development only and would be disabled for production.
- `cors()`: as the server is running a Node instance alongside Tomcat CORS must be enabled in order to be able to serve the requests.
- `exceptionHandling().authenticationEntryPoint(unauthorizedHandler)`: by default Spring returns a 401 with a predefined message, however, a custom `AuthenticationEntryPoint` class is implemented so a more specific message can be sent back to the client.
- `.authorizeRequests().antMatchers("/","/ws/**")`: This allows all requests that are to the base path "/" or to where the web socket is configured at "/ws/**". As the server is serving the client resources, the server must allow requests to "/" as the client uses Redux. When the user clicks on a link and it takes them to a new page, the address bar changes and the server sees this as a request to a forbidden route and forbids it. Allowing all requests to the base allows for the client to function correctly. As there are no controllers serving at the base there is no server side impact.
- `.antMatchers("/api/authorisation/**").permitAll()`: this allows all requests to the `AuthenticationController` in order for a user to query sign up information, sign up and log in. Thus, allowing for their authentication token to be served and then access protected routes. This configuration notifies the application that requests are allowed with no currently registered `UserPrincipal`.
- `.antMatchers("/api/**").authenticated()`: this sets the security configuration to block all other requests that did not match a previous setting. Now the application is fully secure and only authorised requests can pass.
- `http.addFilterBefore(authenticationRequestFilter(), UsernamePasswordAuthenticationFilter.class)`: registers the `AuthenticationRequestFilter` that will be used to verify the JWT tokens.

IV. Client Implementation

The client application is built using React.JS and features a full what you see is what you get (WYSIWYG) editor that users can use to take notes and format them. The client has been designed to be as familiar in it's appearance as other text editors and to be easy to navigate. The application uses several interesting technologies:

- **React Native**: is used for mobile devices, however, there is a port for the web that allows these technologies to be used there. As such, most of the client code also scales to mobile devices; though there are some improvements that need to be made at present.
- **Redux Router**:. Redux is a central state container for JavaScript based applications. Redux Router is a framework that takes care of navigational components and this allows for the ease of navigating between pages and loading the state required for that page. This results in pages that can be bookmarked, refreshed and ones that can be easily navigated to. In addition to this, if a user attempts to navigate to a page that requires authentication and they are not logged in then they are navigated to the login page.
- **Ant Design**: for a large portion all of the UI components. All of their components integrate well and they feature brilliant support for their project.
- **React Quill**: is a React version of QuillJS and is used for the WYSIWYG editor. The editor was configured as it's own component and then used in the notes page itself.

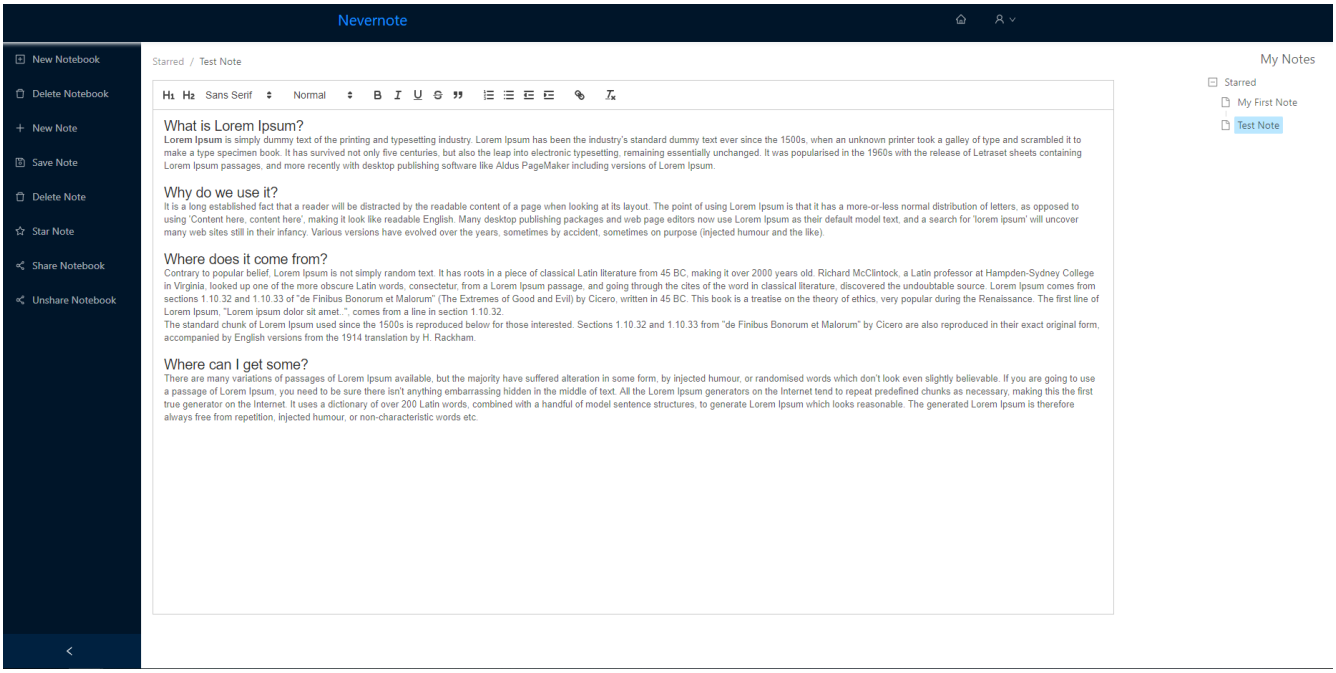

Fig. 6: React JS Client Application

A. Integration with Maven

The source code for the client is available in `/src/main/ui`. When the server is run, there is a Maven task that downloads a Node instance and calls `npm start` which pulls all of the required dependencies and starts the Node server; this functionality is achieved through the `frontend-maven-plugin` and a custom build plugin configuration.

Underneath this, this task also runs the Babel compiler and Webpack. This takes all of the client resources and bundles it in to a single `App.js` file alongside a static HTML file that includes it and puts them to the `/src/main/resources/static/` directory allowing Tomcat to serve a single file. As a result of the security configuration, this results in the UI being served at any path and if no available route is configured in Redux Router then a 404 page is served.

Alongside this, the client development environment has been configured (`webpack.config.js`) to be hot swappable to speed up development. If the node server is started outside of the Tomcat server then it starts and runs a proxy to port 8080 from 9090, automatically. This, however, requires running the web server by itself.

B. Design

The client has largely been developed with the text area to be the main focus and all required features placed around it. The left pane features functions that affect notes and notebooks and the right pane features a tree of the user's notebooks and notes within; the tree's leaves are the notes themselves; see figure 9.

When a user attempts to create a notebook, the request is sent to the server and a validator is run and if it fails the exception message is mapped back to the client. The client recognises that a HTTP 400 Bad Request was returned and automatically creates a notification that displays the validation message to the client; see figure 8.

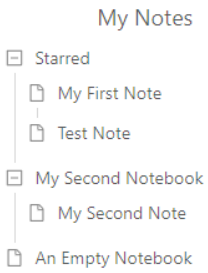

Fig. 7: Tree of Notebooks and Notes

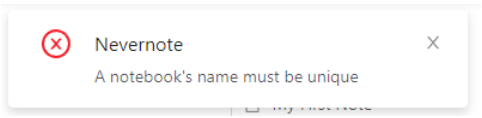

Fig. 8: Error when attempting to duplicate a notebook name

Following the user using the functions in the drawer on the left of the UI, the window refreshes and updates with the new information; this is something that should be changed in the future as it is not the most user friendly approach to this functionality. The user has total control over their notes and notebooks with the exception of being able to delete the Starred notebook, as they cannot; starring a note moves it to this notebook and it must exist. Users can also choose to share a notebook with another user but they must know the user's email address, though this is something that could be improved upon at a later date through the addition of a friends function or through a search facility. If a user wishes to unshare a notebook they can choose to do so too.

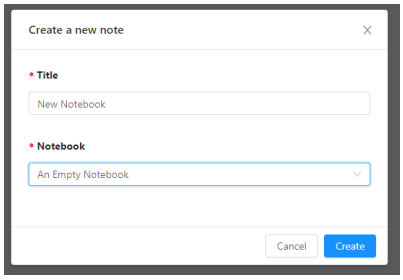

Fig. 9: New note modal

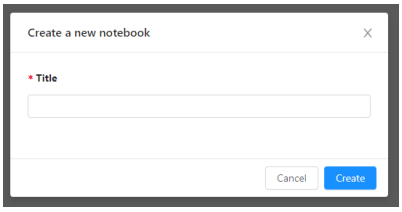

Fig. 10: New notebook modal

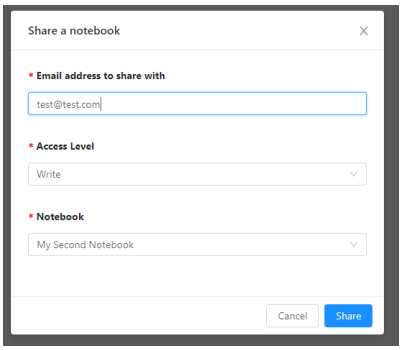

Fig. 11: Share notebook modal

C. Implementation

The application is designed such that each `.JS` file contains it's own class and this contains it's own unique functionality. The classes then build up the views that the user can use and they are contained within a `App.JS` file that determines what view to deliver to the user based on their requested URL. This design has allowed for more rapid development and lends itself to being maintainable in the future. For example, the current page can be changed with a function call as simple as `this.props.history.push("/notes/")`.

Through this design a number of reusable components have been developed; such as the modal window system, populating the note list and comboboxes that rely on a list of notes. Such as, quickly being able to determine whether or not a note is editable based on permissions is just passing a command to the editor component when rendering. This dynamic functionality only affirms that React was the correct choice for this application; as it is state driven.

All API calls go through a single function that attaches the Authorization header, if it exists, to the API call and then updates the one in the browser's local storage with the updated token; this removes duplicate code throughout the application. Other than the user's token, no other details need to be sent along with the request as the context of the request can be retrieved from the token.

The user has the ability to log out of the application and when they choose to do so the local storage is cleared of the access token so that an expired one is not sent to the server and their request is rejected.

V. Notifications

A. Server

When investigating notification systems, several were considered before a strategy was set upon. Requirements were made that the system needed to be scalable, allow for notifications to wait for the user to log in and allow for separation of concerns.

- **Firebase Cloud Messaging**: FCM [15] is a powerful messaging service that is integrated in to the Firebase platform and immediately offers itself as a strong competitor to any other solution, however, in this instance it is overkill for what is required. While it would offer itself as a long-term scalable solution, messaging queues and sockets do too. FCM would not work on a solution that is to be deployed in-house.
- **RabbitMQ and Socket.IO**: Implementing a messaging service is the best approach for this feature and RabbitMQ coupled with Socket.IO [16] is an acceptable solution. Since the client is run using a Node server and this features an AMQP client using Socket.IO would be a solution.
- **Spring WebSockets**: using the Spring ecosystem has many benefits and the addition of websockets is one. Native support that easily integrates in to a project with simple setup is a strong competitor to the above options.

Given that Spring offers websockets it is the best option to implement a notification system in to the application. Spring WebSockets [17] are implemented using the Streaming Text Oriented Messaging Protocol (STOMP) and easily integrate in to the project through the use of Spring's `SimpMessagingTemplate` class. Once this class is instantiated, a messaging service is created and all that is required to send a message to a user is setting the destination to the user's username and for a client to be listening to this queue. The server side implementation of this is:

```
void notifyUser(User user, String message) {
   messagingTemplate.convertAndSendToUser(user.getUsername(), "/topics/all", message);
}
```

B. Client

Client side messaging was also developed using STOMP through the node package `react-stomp` [18]. Due to the way React is designed (all components in a displayed class are live, but just not visible), the messaging service must be built in to the header as it is always displayed. However, the user may not always be logged in, so the topic channel is not registered until the user is actually authenticated and the username is set locally. Once the user is authenticated, the client subscribes to it's topic. This is done as follows:

```
<SockJsClient
   url='http://localhost:8080/ws'
   topics={["/user/" + (this.props.currentUser && this.props.currentUser.username) + "/topics/all"]}
   onMessage={(msg) => {
      notification.info({
         message: 'Nevernote',
         description: msg
      })
      }}
   ref={ (client) => { this.clientRef = client }}
/>
```

Notice the URL is what was previously configured as acceptable in the security configuration. When a user has a notebook shared with them they receive a message similar to figure 12.

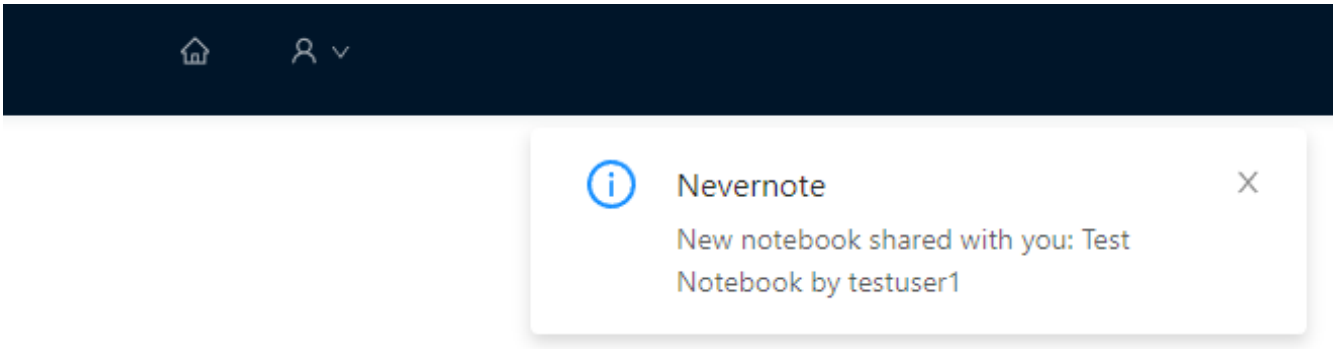

Fig. 12: Notification via WebSocket

C. Improving the system

While the implementation does as required, it does lend itself to some areas for improvement at a later date. If the server was to restart then all active messages would be lost and the user would not know about any changes. The server could add all messages pending delivery to the database and once they have delivered then they could be removed. This change that would improve the user experience of the application significantly. In addition to this, if the user was not available to accept delivery of a message, then the next time they login all pending messages could go to a notification drawer and they could read them all there and they could then be marked as delivered.

The use of WebSockets for the notification system results in a scalable solution that would also work well on mobile devices. If the system moved to storing pending notifications in a database then a push notification could also be sent to the user's mobile device.

REFERENCES

[1] "React: A javascript library for building user interfaces." [Online]. Available: https://reactjs.org/
[2] "Redux - a predictable state container for js apps." [Online]. Available: https://redux.js.org/
[3] "Babel the compiler for next generation javascript." [Online]. Available: https://babeljs.io/
[4] "bundle your assets scripts." [Online]. Available: https://webpack.js.org/
[5] "Spring boot." [Online]. Available: https://spring.io/projects/spring-boot
[6] "Spring security." [Online]. Available: https://spring.io/projects/spring-security
[7] "Spring data." [Online]. Available: https://spring.io/projects/spring-data
[8] "H2 database engine." [Online]. Available: http://www.h2database.com/
[9] "How to fix sql injection: Jpa." [Online]. Available: https://software-security.sans.org/developer-how-to/fix-sql-injection-in-java-persistence-api-jpa
[10] "Hibernate orm." [Online]. Available: http://hibernate.org/orm/
[11] Jan 2013. [Online]. Available: https://docs.oracle.com/javaee/6/tutorial/doc/gipmb.html
[12] "Cross-origin resource sharing (cors)." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
[13] Spring-Projects, "Sec-3030: Reduce in memory lifetime of passwords: Issue 3238. spring-projects/spring-security." [Online]. Available: https://github.com/spring-projects/spring-security/issues/3238
[14] Jwtk, "jwtk/jjwt," Feb 2019. [Online]. Available: https://github.com/jwtk/jjwt
[15] "Firebase cloud messaging | firebase." [Online]. Available: https://firebase.google.com/docs/cloud-messaging/
[16] "rabbitmq node.js = rabbit.js." [Online]. Available: https://www.rabbitmq.com/blog/2010/11/12/rabbitmq-nodejs-rabbitjs/
[17] [Online]. Available: https://docs.spring.io/spring/docs/5.0.0.BUILD-SNAPSHOT/spring-framework-reference/html/websocket.html
[18] Lahsivjar, "lahsivjar/react-stomp," Feb 2019. [Online]. Available: https://github.com/lahsivjar/react-stomp

End of Examination