### Front-end data interaction

Because our project adopts the design idea of front and rear end separation, the technology of front and rear end data interaction is involved in many business logic implementations.

##### Main application scenarios should be

1. User information is requested from the back end when the user logs in
2. When applying for policy compensation, the form information filled by the user is sent to the back end
3. When a user views his or her own policy application list, the user application data is retrieved from the back end
4. When the employee views the list of customer application forms, it gets the application data of all users from the back end
5. The data is sent to the back end when the user consults the problem
6. When employees reply to customer questions, they get the customer inquiry list from the back end

*The above is only the main application scenario, and more front-end and front-end data interaction scenarios are involved in the actual application*

##### Concrete examples

Next, I will take the user's application for policy compensation as an example (main application scenario 2) to describe the implementation of front and rear end interaction in detail.

* When the user fills in the application form, click "ok", the content filled by the user will be first encapsulated in the formdata class, and the variable values in the class will be formatted and validated with the $refs[].validate() method. If the validation fails, the error message will be returned to the user and the user will fill in again. You can see the data structure for formdata class in the appendix[N](ER).
* After the validation passes, the data is sent to the back end using **axios**().Because the form contains user sensitive information, the **post** method is used here, and a **url**(/form) is specified, the formdata class is sent to the back end, and axios automatically encapsulates the data as a **json** type.

Enclosing axios ({  
Method: "post",  
Url: "/ form",  
Data: enclosing formdata  
})

* In addition, the agent needs to be set in the configuration file of vue. Config. js to achieve cross-domain access. Thus, the front-end task is completed.

Proxy: {  
"/ interest" : {  
Target: "http://localhost:8080",  
Secure: false  
}

* The back end receives the data sent by the front end using the @postmapping ("/form") annotation in the FormController class and implements the business logic processing in its annotated methods.

### Back-end business logic processing

After realizing the data interaction between the front end and the back end, it is necessary to make a correct response to the data from the front end. The realization is mainly in the service layer, including the judgment and processing of business logic, and the standardized processing of business results will be returned

#### The primary business logic implementation

Most of the business logic is received by the controller class data and requests, and through the implementation of service interface serviceImpl class to deal with the specific business logic, and finally through the Dao layer to achieve the interaction with the database.

Controller - > service - > serviceImpl > Dao - > SQL

The following examples illustrate specific implementations of several major business logic

##### User registration

* In the UserController class, first @postmapping () receives the url for the newly created user information ("/roles/ roles ") and extracts the user information from the RoleEntity class with the @requesrbody annotation, then stores the user data using the insertRole() method in the RoleService interface

@ PostMapping ("/roles/role ")  
Public ResponseWrapper<RoleEntity> insertRole(@requestbody RoleEntity RoleEntity) {  
RoleService. InsertRole (roleEntity);  
The log. The debug (" The method is ending ");  
Return new ResponseWrapper < > (roleEntity);  
}

* The RoleServiceImpl class will implement the RoleService interface and implement the insertUser() method to store the data, where we use Bcrypt to encrypt the user password to protect the user's information security

@ Override  
Public void insertUser(UserEntity) {  
UserEntity. SetPassword (" {bcrypt} "+ new BCryptPasswordEncoder () encode (userEntity. GetPassword ()));  
UserDao. InsertUser (userEntity);  
}

##### Displays the list of forms requested by the logged-in user

* In the FormController class, get the number of pageSize and the current page number of the form displayed per page from the data transmitted from the front end
* Get the ID of the current user through the getId() method in the SecurityAuthenUtil class to retrieve all forms for that user's application.
* Create a new entity of the PageResult class to encapsulate the list of forms to be displayed and the number of all forms to be displayed on the current page returned from the database, and get the values of these two variables through the formList and formsSize in the FormService interface.

@ ApiModelProperty (" the paging data ")  
Private List <?> data;  
​  
@ ApiModelProperty (" total paging data ")  
Private Integer totalCount;

* The resulting PageResult entity is returned

Return new ResponseWrapper < > (pageResult);

### Database layer interaction

In order to separate the business logic from the database access logic, we use the persistence layer framework MyBatis, which enables the Dao layer to realize the interaction function with the database in addition to the function of adding, deleting, modifying and checking data. You can see the ER diagram of the database in the appendix[N](ER).

#### The specific implementation

* First, all Dao layer classes are annotated with the @mapper annotation, which turns the original class into an interface. And @mapper generates implementation classes for these interfaces that are referenced by other classes.
* In these Dao interfaces, we add the required methods to add, delete, modify and check the data
* Later, we will correspond to a \*\*Mapper. XML file for each Dao interface, in which only SQL statements return data requested by each method in Dao interface.

*I'll pick one of several implementations in the back-end business logic processing for illustration below*

##### Displays the list of forms requested by the logged-in user

* In the Regis\_FormDao interface, there is a formList and a formsSize method, which pass the required data from the SQL statement to the Regis\_FormMapper. XML file through @param

List<Regis\_FormEntity> formList(@param ("pageSize") int pageSize, @param ("start") int start, @param ("ID") int ID);  
​  
Integer formsSize(@param ("pageSize") int pageSize, @param ("start") int start);

* In the Regis\_FormMapper. XML file, the method in the Regis\_FormDao interface is corresponding to the id in the select component,resultMap is used to define the type of returned data, and SQL statement is implemented in this component.

< select id = "formList" resultMap = "formModelMap" > < / select >

### User password encryption

To protect the user's privacy, we encrypt the user's password through Bcrypt before storing it to the database

UserEntity. SetPassword (" {bcrypt} "+ new BCryptPasswordEncoder () encode (userEntity. GetPassword ()));

### The user authorization

* Our application is divided into client side and employee side, so we need to configure reasonable permissions for each role, and we adopt the Spring security framework. Through inheritance ResourceServerConfigurerAdapter class and overriding its methods configure () for each role assigned permissions.
* For example, clients can access all functions whose urls are public/\*\*

HTTP. AuthorizeRequests (.) antMatchers ("/public / \* \* "). PermitAll ()

* The employee can access not only the url that the employee can access, but also the url requests such as /admin/\*\*

HTTP. AuthorizeRequests (.) antMatchers ("/roles / \* \* ", "/ admin / \* \*"). The hasRole (" admin ")