

Golang - III assignment

INFT4245 Golang application development

01.11.2024

Kuznetsov Maksim

1#Introduction

In this project, I will implement a basic backend in Go to work with RESTful APIs using the Gin framework. The backend serves as the basis for interacting with the frontend application built on React and Vite. I'm also using the axios library to easily manage client-side API requests, and the request state is handled with a custom useApi hook. The main goal is to keep the code simple and clear, and to improve the efficiency of interaction between frontend and backend.

2#Connecting with Frontend, Part 1

To get started, we need to implement a basic RESTful API using Go and the Gin framework. With the framework, you can implement a minimal version of the backend quickly and efficiently. I use a local variable to store data to simplify communication in roots.

```
var books = []m.Book{
    {ID: 1, Title: "Book One", Author: "Author One", Description: "First book description"},
    {ID: 2, Title: "Book Two", Author: "Author Two", Description: "Second book description"},
}
var nextID = 3

func SetupRoutes(router *gin.Engine) {
    router.POST("/books", createBook)
    router.GET("/books", getBooks)
    router.GET("/books/:id", getBookByID)
    router.PUT("/books/:id", updateBook)
    router.DELETE("/books/:id", deleteBook)
}
```

In the internal/books file, I have declared a SetupRoutes function with set up all the Routes for CRUD operations with the Book structure

```
type Book struct {
    ID          int    `json:"id"`
    Title       string `json:"title"`
    Author      string `json:"author"`
    Description string `json:"description"`
}
```

As a result, when you perform a GET request /books, the result comes from a local array:

```
← → ↻ ⓘ http://localhost:8080/books
Автоформатировать ☒
[
  {
    "id": 1,
    "title": "Book One",
    "author": "Author One",
    "description": "First book description"
  },
  {
    "id": 2,
    "title": "Book Two",
    "author": "Author Two",
    "description": "Second book description"
  },
]
```

To initialize the React application, I used Vite, a local server for frontend development using Typescript for typing and plugged in the tailwind library for quick component styling:

```
(use --node --trace-warnings ... to show where the warning was created)
✓ Project name: ... assignment3-client
✓ Select a framework: » React
✓ Select a variant: » TypeScript

Scaffolding project in D:\Study\4 course\gocloudload-4-course\Golang\3\tmp\assignment3-client...

Done. Now run:

  cd assignment3-client
  npm install
  npm run dev

onlyk@Only-Yoga-7 MINGW64 /d/Study/4 course/gocloudload-4-course/Golang/3/tmp (main)
```

To display the list of books, I made a simple HomePage and displayed it through Router:

```
const HomePage = () => {
  const { data: books } = useApi<Book[]>('books', { default: [] });

  return (
    <Layout>
      <div className="text-2xl font-bold text-default-700">All Library</div>

      <div className="grid grid-cols-4 gap-2 mt-4">
        {books.map((book) => (
          <div className="p-4 bg-content2 rounded-2xl">
            <div className="text-tiny opacity-50">{book.id}</div>
            <div className="text-medium my-1">{book.title}</div>
            <div className="text-small text-default-500 line-clamp-2">{book.description}</div>
          </div>
        ))}
      </div>
    </Layout>
  );
};
export default HomePage;
```

All Library

- 1

Book One

First book description
- 2

Book Two

Second book description
- 3

fawefawef

To work with the API server, I implemented a hook with axios based states:

/hooks/useApi.ts

```
const api = axios.create({ baseUrl: 'http://localhost:8080/' });

export const useApi = <T>({url: string, props: ApiProps = {}} => {
  const [data, setData] = useState<T>({props.default || null});
  const [loading, setLoading] = useState(props.initLoading || false);
  const [error, setError] = useState<string>('');

  useEffect(() => {
    (async () => {
      setError('');
      setLoading(true);
      try {
        let res;

        switch (props.method) {
          case 'POST':
            res = await api.post(url, props.body || {}, props.config);
            break;

          case 'PATCH':
            res = await api.patch(url, props.body || {}, props.config);
            break;

          case 'DELETE':
            res = await api.delete(url, props.config);
            break;

          default:
            res = await api.get(url);
            break;
        }

        setData(res.data);
      } catch (e: any) {
        if (isAxiosError(e)) {
          setError(e.message);
        } else {
          setError(e.message || 'Unknown Error');
        }
        console.error(e);
      } finally {
        setLoading(false);
      }
    })();
  }, [url]);

  return { data, loading, error };
});
```

When loading a component with this hook, thanks to `useEffect`, a request is sent to the server, all stages of the request (loading, success, error) are written to states and returned from the hook. Thus, the readability of code and logic is improved.

3#Connecting with Frontend, Part 2

In the second part, I added new pages to work with the API:

```

client > src > App.tsx > App
1  import { Navigate, Route, Routes } from 'react-router-dom';
2
3  import CreatePage from './pages/Create';
4  import DetailsPage from './pages/Details';
5  import HomePage from './pages/home';
6
7  function App() {
8      return (
9          <Routes>
10             <Route element={<HomePage />} path="/" />
11             <Route element={<DetailsPage />} path="/:id" />
12             <Route element={<CreatePage />} path="/create" />
13             <Route path="/*" element={<Navigate to="/" />} />
14         </Routes>
15     );
16 }
17
18 export default App;
19

```

The DetailsPage makes a request to get the book by id, and after loading it displays the data. And on the CreatePage page I implemented a form for creating a new book and calling POST request with data from the form.

For error handling, I used the error state of the useApi hook, which I described earlier, and output the error if there is one:

```

return (
    <Layout>
        <button className="p-2 rounded-lg bg-content3" onClick={() => navigate(-1)}>
            Back
        </button>

        <div className="text-2xl font-bold text-default-700">Create New Book</div>
        <div className="flex flex-col gap-3 w-64 mt-4">...
        </div>

        {error && <div className="p-2 rounded-lg border-danger">{error}</div>}
    </Layout>
);
};

```

http://localhost:5173/create

[Back](#)

Create New Book

New Book

Description

Author

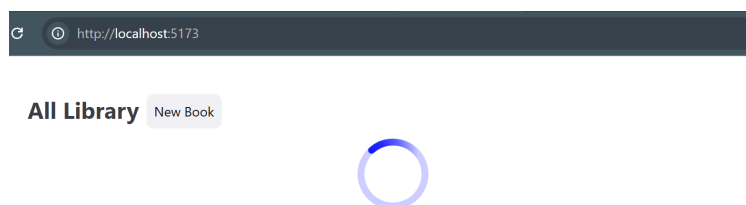
Create

Description is required

To display the loading status, I created a new `<Loader />` component, which consists of an animated svg component I took from www.svgbackgrounds.com. On the HomePage, I use the state of the useApi hook - loading, to track the progress of the request:

```
5
6 < const HomePage = () => {
7   const navigate = useNavigate();
8   const { data: books, loading } = useApi<Book[]>('books', { default: [] });
9
10  return (
11    <Layout>
12      <div className="flex gap-2">
13        <div className="text-2xl font-bold text-default-700">All Library</div>
14        <button
15          className="p-2 text-small rounded-lg bg-content2 transition-all hover:bg-content3
16          onClick={() => navigate('/create')}
17        >
18          New Book
19        </button>
20      </div>
21
22      {loading && <Loader />}
```

The loading status helps the user understand what is happening on the site, and adds better responsiveness to the interface.



To add new books, I created a new CreatePage, with a form to fill in the data.

```

const CreatePage = () => {
  const navigate = useNavigate();
  const [form, setForm] = useState<Omit<Book, 'id'>>({
    title: 'New Book',
    description: '',
    author: '',
  });
  const {
    data: newBook,
    loading,
    fetch,
    error,
  } = useApi<Book>(`books`, { method: 'POST', manual: true, body: form });

  const onChangeHandler = (e: any) => {
    setForm({ ...form, [e.target.name]: e.target.value });
  };

  useEffect(() => {
    if (newBook) {
      navigate(`/${newBook.id}`);
    }
  }, [newBook]);
}

```

After successful creation, the user is redirected to the DetailsPage. And during the request execution, the button is replaced by an inactive download block:

Back

Create New Book

New Book

Description

Author

Creating...

For more efficient management of the main states, I created a global React context - AppContext, into it I moved the useApi hook to call the list of all books, it is also executed on page load and stored globally. Thanks to this structure it is possible to reduce the number of requests when navigating to the HomePage, because the data will already be loaded into the global context at the first load.

```

export const AppProvider = ({ children }: any) => {
  const [user, setUser] = useState<User | null>(null);
  const {
    loading: booksLoading,
    data: books,
    error: booksError,
  } = useApi<Book[]>('books', { default: [] });

  const updateUser = (user: User) => {
    setUser(user);
  };

  return (
    <AppContext.Provider
      value={{
        user,
        updateUser,

        books,
        booksLoading,
        booksError,
      }}
    >
      {children}
    </AppContext.Provider>
  );
};

```

To make it easier to use the context in other components, I declared a useApp hook that returns an instance of the current context.

And after a little refactoring of the main page, I reduced the amount of code for accessing the server API:

```

const HomePage = () => {
  const navigate = useNavigate();
  const { books, booksLoading } = useApp();

  return (
    <Layout>
      <div className="flex gap-2">
        <div className="text-2xl font-bold text-default-700">All Library</div>
        <button
          className="p-2 text-small rounded-lg bg-content2 transition-all hover:bg-content3"
          onClick={() => navigate('/create')}
        >
          New Book
        </button>
      </div>

      {booksLoading && <Loader />}

      <div className="grid grid-cols-4 gap-2 mt-4">
        {books.map((book) => (
          <div key={book.id} className="p-4 bg-content2 rounded-2xl">
            <div className="text-tiny opacity-50">{book.id}</div>

```

3#Authentication and Authorization

To add authorization using the JWT token, additional dependencies must be installed:

go get github.com/golang-jwt/jwt/v4

To create a JWT token, I implemented the generateJWT method, which uses the NewWithClaims function that encrypts the data based on the HS256 encryption

algorithm and the JwtSecret key. In the JWT token, I store username data for further user experience and exp field to track the expiration date of the token.

```
1  package user
2
3  import (
4      "time"
5      "github.com/golang-jwt/jwt/v4"
6  )
7
8  var JwtSecret = []byte("SECRET_KEY")
9
10 func generateJWT(username string) (string, error) {
11     token := jwt.NewWithClaims(jwt.SigningMethodHS256, jwt.MapClaims{
12         "username": username,
13         "exp":      time.Now().Add(time.Hour * 72).Unix(),
14     })
15     return token.SignedString(JwtSecret)
16 }
17
```

Для использования авторизации, я также создал структуру пользователя и методы для авторизации и регистрации:

```
:GO service.go book U   :GO auth.go user U   :GO models.go internal U X   :GO
backend > internal > :GO models.go > ...
1  package internal
2
3  type User struct {
4      ID          int    `json:"id"`
5      Username    string `json:"username"`
6      Password    string `json:"password"`
7  }
8
9  type Book struct {
10     ID          int    `json:"id"`
11     Title       string `json:"title"`
12     Author      string `json:"author"`
13     Description string `json:"description"`
14 }
15
```

When registering a user, I check if the user exists in the array. To encrypt the password, I use the bcrypt library and the GenerateFromPassword method. As a result, if the request is successful, the created user and the generated token are returned:


```

func registerUser(c *gin.Context) {
    var newUser m.User
    if err := c.ShouldBindJSON(&newUser); err != nil {
        c.JSON(http.StatusBadRequest, gin.H{"message": err.Error()})
        return
    }

    if userExists(users, newUser.Username) {
        c.JSON(http.StatusBadRequest, gin.H{"message": "User already exist"})
        return
    }

    hashedPassword, err := bcrypt.GenerateFromPassword([]byte(newUser.Password), bcrypt.DefaultCost)
    if err != nil {
        c.JSON(http.StatusInternalServerError, gin.H{"message": "Failed to hash password"})
        return
    }
    newUser.Password = string(hashedPassword)
    newUser.ID = nextUserID
    nextUserID++

    users = append(users, newUser)

    token, err := generateJWT(newUser.Username)
    if err != nil {
        c.JSON(http.StatusInternalServerError, gin.H{"message": "Failed to generate token"})
        return
    }

    c.JSON(http.StatusCreated, gin.H{"token": token, "user": newUser})
}

```

In the authorization method, the user's presence in the database and a password match are checked via bcrypt methods. At the end user and token are also returned.

```

func loginUser(c *gin.Context) {
    var creds m.User
    if err := c.ShouldBindJSON(&creds); err != nil {
        c.JSON(http.StatusBadRequest, gin.H{"message": err.Error()})
        return
    }

    var user m.User
    for _, u := range users {
        if u.Username == creds.Username {
            user = u
            break
        }
    }

    if user.ID == 0 || bcrypt.CompareHashAndPassword([]byte(user.Password), []byte(creds.Password)) != nil {
        c.JSON(http.StatusUnauthorized, gin.H{"message": "Invalid username or password"})
        return
    }

    token, err := generateJWT(user.Username)
    if err != nil {
        c.JSON(http.StatusInternalServerError, gin.H{"message": "Failed to generate token"})
        return
    }

    c.JSON(http.StatusOK, gin.H{"token": token, "user": user})
}

```

To test how authorization works, I added the AuthMiddleware method to handle authorized requests. It extracts a Bearer Token from the Authorization header, which is parsed using methods from the jwt-go library. There are several checks for validity

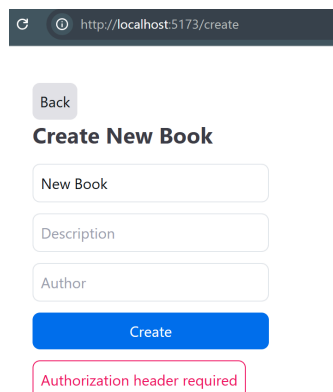
and presence of the token. Also the data from the token is stored locally in the Gin context of the request.

```
12 func AuthMiddleware(c *gin.Context) {
13     authHeader := c.GetHeader("Authorization")
14     if authHeader == "" {
15         c.JSON(http.StatusUnauthorized, gin.H{"error": "Authorization header required"})
16         c.Abort()
17         return
18     }
19
20     tokenString := strings.TrimPrefix(authHeader, "Bearer ")
21     token, err := jwt.Parse(tokenString, func(token *jwt.Token) (interface{}, error) {
22         if _, ok := token.Method.(*jwt.SigningMethodHMAC); !ok {
23             return nil, jwt.NewValidationError("Unexpected signing method", jwt.ValidationErrorSignatureInvalid)
24         }
25         return JwtSecret, nil
26     })
27
28     if err != nil || !token.Valid {
29         c.JSON(http.StatusUnauthorized, gin.H{"error": "Invalid token"})
30         c.Abort()
31         return
32     }
33
34     claims, ok := token.Claims.(jwt.MapClaims)
35     if !ok || !token.Valid {
36         c.JSON(http.StatusUnauthorized, gin.H{"error": "Invalid token claims"})
37         c.Abort()
38         return
39     }
40
41     c.Set("username", claims["username"])
42     c.Next()
43 }
```

To apply the middleware created, I updated the SetupRoutes method in the package

```
18 func SetupRoutes(router *gin.Engine) {
19     router.GET("/books", getBooks)
20     router.GET("/books/:id", getBookByID)
21
22
23     router.POST("/books", m.AuthMiddleware, createBook)
24     router.PUT("/books/:id", m.AuthMiddleware, updateBook)
25     router.DELETE("/books/:id", m.AuthMiddleware, deleteBook)
26 }
```

book:



Back

Create New Book

New Book

Description

Author

Create

Authorization header required

To connect authorization on the client, I added new methods to the global context and created a new page with a form for authorization.

```

20 export const AppProvider = ({ children }: any) => {
21   const [token, setToken] = useState<string>('');
22   const [user, setUser] = useState<User | null>(null);
23   const {
24     loading: booksLoading,
25     data: books,
26     error: booksError,
27   } = useApi<Book[]>('books', { default: [] });
28
29   const updateUser = (user: User) => {
30     setUser(user);
31   };
32
33   const login = (user: User, token: string) => {
34     setUser(user);
35     setToken(token);
36     localStorage.setItem('token', token);
37   };
38
39   const logout = () => {
40     setUser(null);
41     setToken('');
42     localStorage.removeItem('token');
43   };
44

```

After successful authorization and retrieval of user data with the token, the login method is called to store the token in localStorage, and the logout method is called to clear authorization states and remove the token from memory.

```

6 const LoginPage = () => {
7   const [form, setForm] = useState<Omit<User, 'id'>>({
8     username: '',
9     password: '',
10  });
11   const { data, loading, fetch, error } = useApi<{ user: User; token: string }>('login', {
12     method: 'POST',
13     manual: true,
14     body: form,
15  });
16   const { login } = useApp();
17
18   const onChangeHandler = (e: any) => {
19     setForm({ ...form, [e.target.name]: e.target.value });
20   };
21
22   useEffect(() => {
23     if (data) {
24       login(data.user, data.token);
25     }
26   }, [data]);

```

To invoke authorized requests, I added a token check to the useApi hook that will automatically add the token to the headers if there is one.

```

5 export const useApi = <T>(url: string | null, props: ApiProps = {}) => {
6   const [data, setData] = useState<T>(props.default || null);
7   const [loading, setLoading] = useState(props.initLoading || false);
8   const [error, setError] = useState<string>('');
9
10  const fetch = useCallback(async () => {
11    if (!url) return;
12
13    const token = localStorage.getItem('token');
14
15    setError('');
16    setLoading(true);
17    try {
18      let res;
19
20      await delay();
21
22      props.config = {
23        ...props.config,
24        ...(token ? { headers: { Authorization: `Bearer ${token}` } } : {}),
25      };
26
27      switch (props.method) {
28        case 'POST':
29          res = await api.post(url, props.body || {}, props.config);
30          break;
31
32        case 'PATCH':
33          res = await api.patch(url, props.body || {}, props.config);
34          break;
35
36        case 'DELETE':

```

⌂ http://localhost:5173/login

Login

▼ Local storage	app-auth-token	eyJhbGciOiJIUzI1NiIsInR5cCI6Ikp...
http://localhost:...	app-lang	"ru"
▶ Session storage	theme	light
IndexedDB	token	eyJhbGciOiJIUzI1NiIsInR5cCI6Ikp...
▶ Cookies		
Private state tokens		
Interest groups		
▶ Shared storage		

To add roles to the API service on Go, I extended the user structure, and changed the registration method where I create a user with the role “user” by default, and store the role in a JWT token for quick validation in AuthMiddleware

```
3 type User struct {
4     ID          int    `json:"id"`
5     Username    string `json:"username"`
6     Password    string `json:"password"`
7     Role        string `json:"role" // user | admin
8 }
```

```
func generateJWT(user m.User) (string, error) {
    token := jwt.NewWithClaims(jwt.SigningMethodHS256, jwt.MapClaims{
        "username": user.Username,
        "role":     user.Role,
        "exp":      time.Now().Add(time.Hour * 72).Unix(),
    })
    return token.SignedString(m.JwtSecret)
}
```

In AuthMiddleware, I added to the request to save the user role to verify access in routes:

```
claims, ok := token.Claims.(jwt.MapClaims)
if !ok || !token.Valid {
    c.JSON(http.StatusUnauthorized, gin.H{"message": "Invalid token claims"})
    c.Abort()
    return
}

c.Set("username", claims["username"])
c.Set("role", claims["role"])
c.Next()
}
```

For the book creation route I added a check for the “admin” role, if the value does not match the handler will throw an error No access:

```
func createBook(c *gin.Context) {
    role, exists := c.Get("role")
    fmt.Println(role)
    if role != "admin" || !exists {
        c.JSON(http.StatusForbidden, gin.H{"message": "Dont have permissions"})
        return
    }

    var newBook m.Book
    if err := c.ShouldBindJSON(&newBook); err != nil {
        c.JSON(http.StatusBadRequest, gin.H{"message": err.Error()})
        return
    }
}
```

[Back](#)

Create New Book

Create

Dont have permissions

4#Conclusion

As a result, I created an application that integrates the backend on Go with the frontend on React, implementing the basic functions for working with book data: adding, viewing and deleting. Through the use of JWT, security was added with authentication and user authorization with role validation, allowing to restrict access to certain routes. In addition, the addition of global context on the client optimized the number of queries, which improved performance and user experience.