DSy StudiApp

Technical Documentation

Andri Joos - Kaj Habegger - Lukas Domeisen

22.05.2023

Contents

1 Abstract

Our project idea is based on our project from SE Project. In SE Project we created a native mobile application for students. This native mobile application gives a student the ability to add semesters, modules aka courses, todos, and other information related to the daily life of a student. In DSy we adapted this idea to a web application which obtains all data from an API. The web application is written in JavaScript with help of the Vue framework. For the API we used Golang. Furthermore, we used Traefik as load balancer, Nginx as web server and Postgres as database management system. A high-level diagram of all components in our project can be seen below.

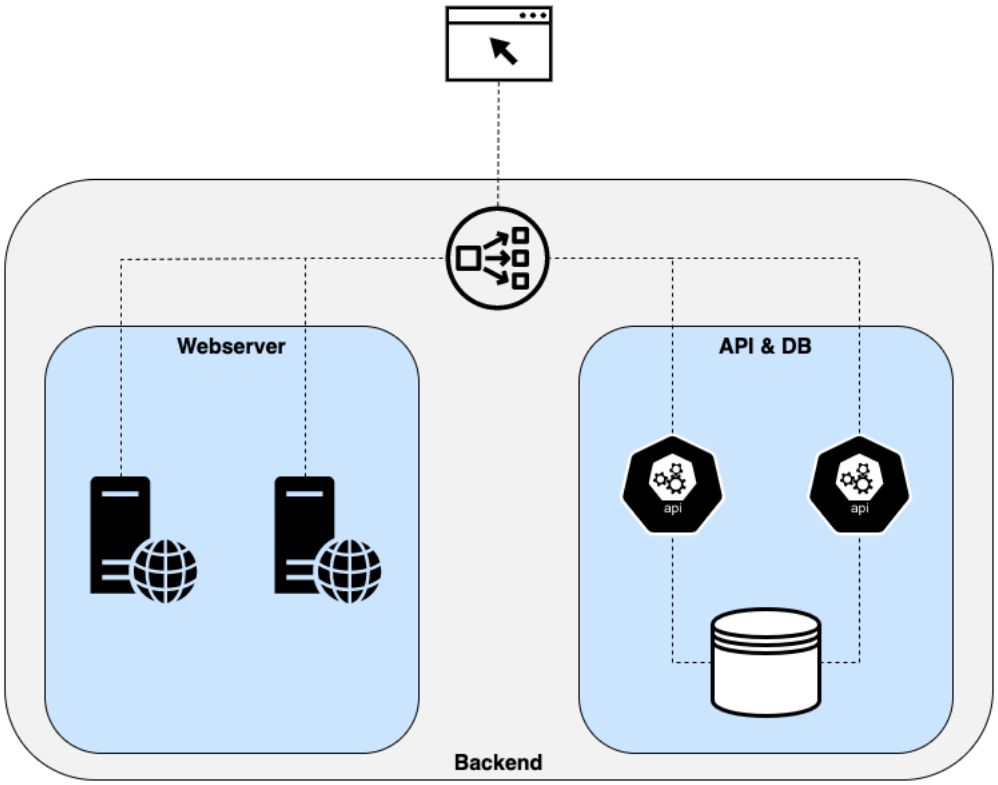

Figure 1: Components

2 Frontend- & API-Development

2.1 Frontend

For the frontend of the web application we used Vue.js (also referred to as Vue), which is an open-source JavaScript framework used for building single-page applications. We also integrated some additional libraries and resources into the Vue application for easier development:

- Vuetify: UI Library for Vue applications containing many good-looking Vue components
- Vue Notifications: Vue library for showing notifications to the user
- Vue Cookies: Vue library for handling browsers cookies
- Axios: HTTP client used for making requests to the backend
- Material design icons: Integrated in Vuetify, used for all icons in the web app

2.1.1 Authentication Token handling

Using an axios interceptor enables us to handle an error response before it gets handled by the actual caller. The following code snippet shows our interceptor implementation for handling cases where the authentication or refresh token has expired.

```
axios.interceptors.response.use(null, error => {
    if (error.response && error.response.config && error.response.status === 401) {
        // 401 error: unauthorized -> refresh token
        return axios.get('auth/refresh', {})
        .then(response => {
            helpers.setLoggedIn(true)
            // And after successful refresh - repeat the original request
            let retryConfig = error.response.config
            return axios.request(retryConfig)
        }).catch(error => {
            helpers.setLoggedIn(false)
            notify('Session has expired')
            // redirect to signin in case refresh request fails
            router.push({ name: 'login' })
            return Promise.reject(error)
        })
    } else if (error.code === 'ECONNABORTED' || !error.response) {
        notify({ title: 'Server is not responding', type: 'error' })
    } else {
        return Promise.reject(error)
    }
})
```

To put it simply, the interceptor does the following:

1. Is error response code 401? (unauthorized error)

   ⇒ Make a token refresh request
   - Success response: Retry the inital request which responded with an unauthorized error (since the token has been refreshed, the request should be successful now)
   - Error response: Redirect to login page and show notification that session expired

2. No response from the server?

   ⇒ Show error notification that server is not responding

3. Else, make the promise fail which will run the catch block of the request caller

2.2 API

Our API, which is responsible for handling the data between the web application and the database, is written in Golang: We used two additional frameworks / libraries to the already given language features of Golang.

- Gin: Is an HTTP web framework written in Golang. We used it to implement the API endpoints which respond to requests of our web application.
- Gorm: Is an ORM library. We used it to interact with the database.

2.2.1 Authentication

Because our web application allows registering, respectively logging in, a user, we built a simple authentication service. Besides register and login it is also possible to log a user out and refresh the session token up to a given time.

- On signup, the password given by the user gets hashed using the established Bcrypt hashing algorithm. Afterwards the user login data will be persisted in the database.
- On a successful login, the user gets an access- as well as a refresh-token in form of cookies.
- On logout, both tokens will become invalid and the user will therefore be logged out.
- On refresh, it will be checked if the refresh token is still valid. If so, the access token will be revalidated.
- With each other request the access token is checked for validity.

The two tokens are JWT tokens. As the service should be as simple as possible for this project it is for example not possible to change a password via the web application.

2.2.2 Architecture

Each information item (e.g. todos, lecture, marks, etc.) is implemented as a data model in our API. These data models consist of a basic struct which shows how the item is structured, structs for each request type and methods for each request / response type, which in turn return instances of structs. Our API controller class consisting of the generic db handler and some data model methods handles the different requests (CRUD operations) for each data model. Furthermore, there is a route controller for each data model which maps the routes onto API controller methods. As soon as the API receives a request on one of the available routes (e.g. GET Request on api/todos/:id) the mapped API controller function will be called. First the called function creates a request object relating to the model (e.g. TodoItemGetRequest). With the created request object the function can now check if the requesting user has permission to get the requested data. If the permission check is successful, the requested resource gets loaded from the database and transformed into a response object (e.g. TodoItemResponse). Then, the response object is sent back to the web application as part of an HTTP response.

Below you will find a graphical interpretation of our API architecture.

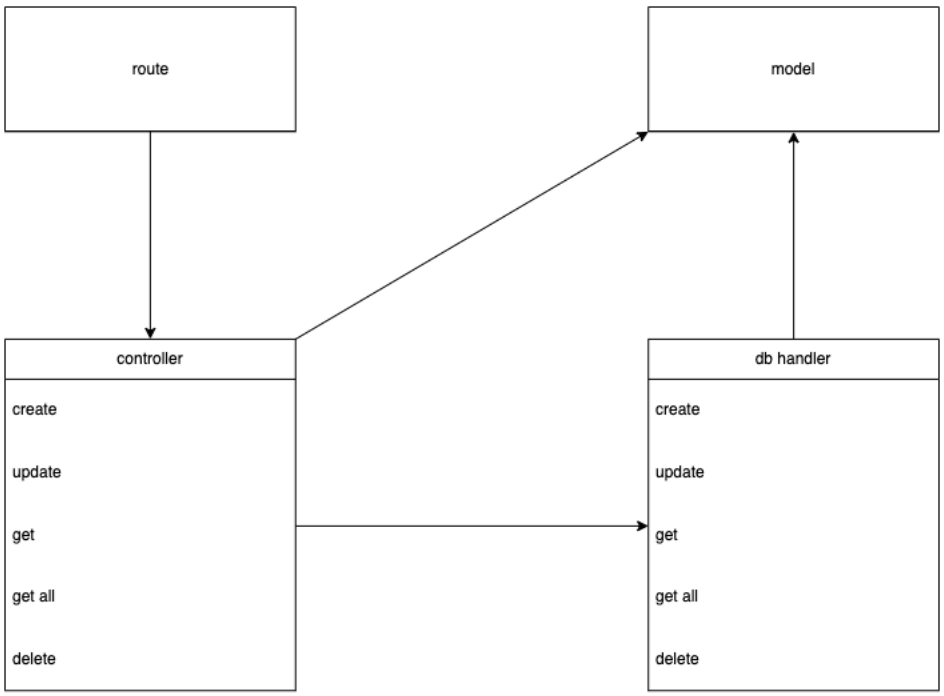

Figure 2: API architecture

Each data model is an implementation of the API model interface. This interface consists of all request / response creation functions. It is important to us to point out, that we used interfaces, because from our point of view Golang and especially Gorm is not meant to work with interfaces. Therefore, it was a real challenge to make this interface work properly.

As already mentioned, generics were used for the db handler. This way we only had to write one db handler for all models. These efforts resulted in low code duplication. As illustrated in the API diagram, the db handler part is an implementation of the repository pattern.

2.2.3 Request example with code snippets

Please note the following code snippets are simplified. The complete code can be viewed having a look at our source code.

1. The web application sends a GET request to api/todos/:id

```
router.GET("/:id", utils.DeserializeUser(), tc.apiController.GetSingle)
```

2. The GetSingle function in the API controller gets called. This method first creates a new GetRequest object, then calls a method from the newly created instance and finally assigns the return value to request.

```
func (ac *ApiController) GetSingle(ctx *gin.Context) {
    request := reflect.New(reflect.TypeOf(ac.GetRequest)).Interface().(
        interfaces.ApiModelGetRequest)

    ...
}
```

3. This is how the GetRequest for a TodoItem looks like

```
type TodoItemGetRequest struct {
    ID uint `uri:"id" binding:"required"`
}
```

4. The following function is called from the previously created instance

```
func (gr TodoItemGetRequest) ToApiModel(user interfaces.UserModel) interfaces.
    ApiModel {
    return TodoItem{
        ID:     gr.ID,
        UserID: user.GetId(),
    }
}
```

5. Back to the GetSingle function, it checks if the user has permission to get the requested information. If the permission check is successful a todo object gets created with the data retrieved from the database.

```
func (ac *ApiController) GetSingle(ctx *gin.Context) {
    ...
    currentUser := ctx.MustGet("currentUser").(interfaces.UserModel)

    ctx.BindUri(request)

    conditions := request.ToApiModel(currentUser)
    model := ac.DbHandler.Get(conditions)

    model = ac.DbHandler.Prefetch(model)
    ...
}
```

6. The following two methods create the final todo object. The Get method creates the object without associated objects. Afterwards the prefetch method adds all associates like the related module aka course object.

```
func (dh DbHandler[T]) Get(conditions interface{}) (interfaces.ApiModel, error
    ) {
    model := new(T)
    err := dh.DB.First(model, conditions).Error
    return *model, err
}

func (dh DbHandler[T]) Prefetch(apiModel interfaces.ApiModel) (interfaces.
    ApiModel, error) {
    model := apiModel.(T)
    err := dh.DB.Model(&model).Preload(clause.Associations).Find(&model).Error
    return model, err
}
```

7. Back to the GetSingle function the final response object gets created by calling the respective GetResponse method. Afterwards it gets attached to the body of the HTTP response.

```
func (ac *ApiController) GetSingle(ctx *gin.Context) {
    ...
    response := model.ToApiModelGetResponse()
    ctx.JSON(http.StatusOK, gin.H{"status": "success", "data": response})
}
```

8. When the following function is called, it returns the response object. Keep in mind that this call for another function is needed to minimize code duplication as other responses exist.

```
func (ti TodoItem) ToApiModelGetResponse() interfaces.ApiModelGetResponse {
        return ti.GetTodoItemResponse()
}

func (ti TodoItem) GetTodoItemResponse() TodoItemResponse {
        return TodoItemResponse{
                ID:       ti.ID,
                Content:  ti.Content,
                Due:      ti.Due,
                ModuleID: ti.ModuleID,
                Finished: ti.Finished,
        }
}
```

3 Backend

3.1 Docker Images

Both docker images are built with docker multi-stage build, so that the number of layers can be kept as low as possible and the docker image to be executed is as small as possible.

3.1.1 API

The first stage of the API image is based on the golang docker image and builds the Golang API. This produces an executable that can be copied into the second stage. This stage is based on the debian image from distroless and runs the previously built executable.

3.1.2 Frontend

Since the frontend is a Vue app, the build stage is based on the node image. The build stage creates a folder containing all scripts necessary to run the website. This folder is then copied to the second stage, which is based on the nginx image and acts as a web server.

3.2 Kubernetes

All services run on an azure kubernetes cluster.

3.2.1 Load Balancer

The team has selected Traefik as Load Balancer since it has HTTP/3 as experimental feature available and no one from the team used it yet. The Load Balancer is available on port 80/TCP for HTTP, 443/TCP for HTTPS and 443/UDP for HTTP/3. To enforce HTTPS however, an HTTP request is always redirected to HTTPS. Additionally, the top-level domain .dev is in the HSTS preload list, so requests over the domain are redirected to HTTPS even before they reach the Load Balancer.

3.2.2 Ingress

Only one ingress is in use. This, because the frontend and the API run in the same namespace and share a domain. While the API responds on the path /api, the frontend responds on the path / but not on /api.

3.2.3 Certificates

Cert-manager in combination with letsencrypt is used to issue a valid SSL certificate. Also, cert-manager ensures, that certificates are renewed before their expiration.

These certificates are used in the ingress to encrypt the HTTP traffic.

3.2.4 Custom Deployments

The API, as well as the frontend, consume the previously built Docker images and are available via the Ingress. Both deployments have two running replicas, so one can immediately take over if the other replica terminates for some reason.

Database
The API needs a database to store the data. The team selected postgres 15 as it is a commonly known database and the team already has some experience with it. To persist the data, a NFS file share, provided by azure, is consumed by the database deployment.

Since Postgres needs to read and write data, only one replica runs at a time. As soon as this instance is terminating, Kubernetes notices this and spins up a new instance.

3.3 DNS

Since the ingress IP differs from the egress IP a DynDNS client, such as ddclient, was not possible. Instead, a static IP was acquired from azure and statically assigned to the domain studiapp.420joos.dev

3.3.1 HTTP/3

HTTP/3 and HTTP/2 are defined as possible protocols in the DNS configuration, with HTTP/3 as default. With this configuration, even the initial request should be HTTP/3 as the browser knows from this DNS entry that the server is able communicate via HTTP/3. Unfortunately, current browsers don't respect this DNS record, so the initial request is currently HTTP/2.

4 Tools

4.1 GitLab

The team uses a selfhosted instance of GitLab for version control and CI/CD operations. This instance is hosted by Andri Joos. It runs on a K3S cluster, which is similar to the one used to run the API and the frontend, but it is runs on a server of Andri Joos.

4.1.1 Agent

To perform actions on the cluster, a GitLab Kubernetes Agent is needed. This agent provides the kubectl context of the cluster.

The agent is installed on the cluster via helm.

4.1.2 Pipelines

The API repository and frontend repository have a pipeline to build their docker image. The build jobs run on a specific GitLab runner that can build docker images. The docker image needed is provided by Andri Joos and is based on docker buildx to build amd64 and arm64 images with only one docker manifest.

The kubernetes repository has a pipeline, which manages the Kubernetes cluster. This pipeline uses the Agent.

Since the documentation is written in LaTeX, the pipeline of the documentation repository builds the documentation into a PDF and publishes it in the artifacts.

4.2 Docker Registry

In order to save the generated docker images, a docker registry was needed. Fortunately, Andri Joos also hosts a docker registry that can be used. This registry uses password authentication, so there is a secret in the cluster that provides the necessary information to authenticate with the registry.

4.3 Local deployment

There is a bash script that helps developing the API. This script

1. builds the docker image of the API
2. if needed, creates a new cluster via k3d, a dockerized version of K3S

3. spins up a NFS server inside the cluster (only Linux, macOS has some issues with permissions, therefore writes directly to the docker volume of the cluster), the NFS docker image is provided by Andri Joos
4. applies a smaller version of the production deployment into the cluster
5. attaches to the pod's logs for debugging.

This ensures compatibility to the production environment.