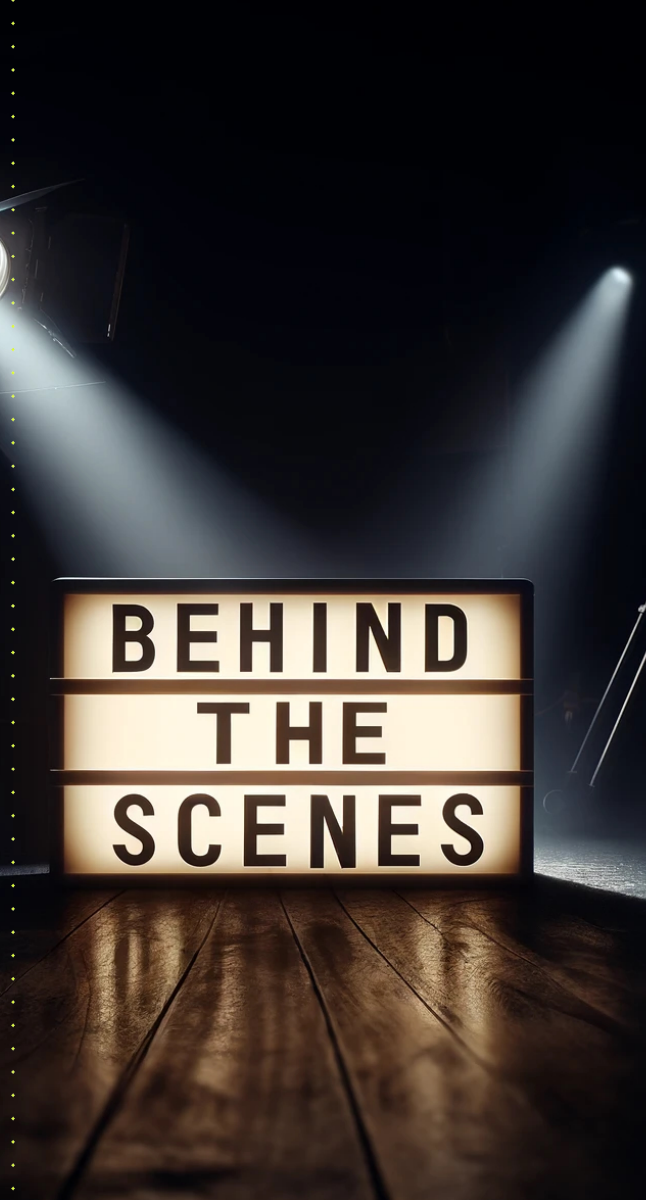
Experiences with Composable Frontends



Miriam Müller

Senior PHP developer (eCommerce)

- Working @sitegeist in Hamburg
- Backend developer in love with composable frontends
- Shopware projects since 2015 (version 4.x)

sitegeist

Agenda

1. Recap Shopware Boost Day #10
2. How to use storybook in composable frontends projects
3. API client: Add own store-api endpoints
4. Integrate B2B components to composable frontends project
5. Conclusion
6. Questions?

1. Recap

Shopware Boost Day #10

sitegeist

Why composable frontends?

- Block overwriting and fighting against stylesheets in complex projects
- Frustrated frontend developers and dissatisfied UX/UI designers
- Project template difficult to control

Design and development process change

- Thinking component first!
- Atomic design and component library
- Component development separated from Shopware integration

Benefits

- No template restrictions
- Clean and independent components
- Free choice of technology
- No in-depth Shopware know-how

2. Storybook

How to use storybook in composable frontends projects

sitegeist

What is storybook?

- Open source tool for component development
- Develop, test and document components without application
- Living styleguide

Integration in composable frontends project

```
/**
 * setup in composable frontends
 * with nuxt3 & vite
 */

npx nuxi@latest module add storybook


npx storybook-nuxt init
```

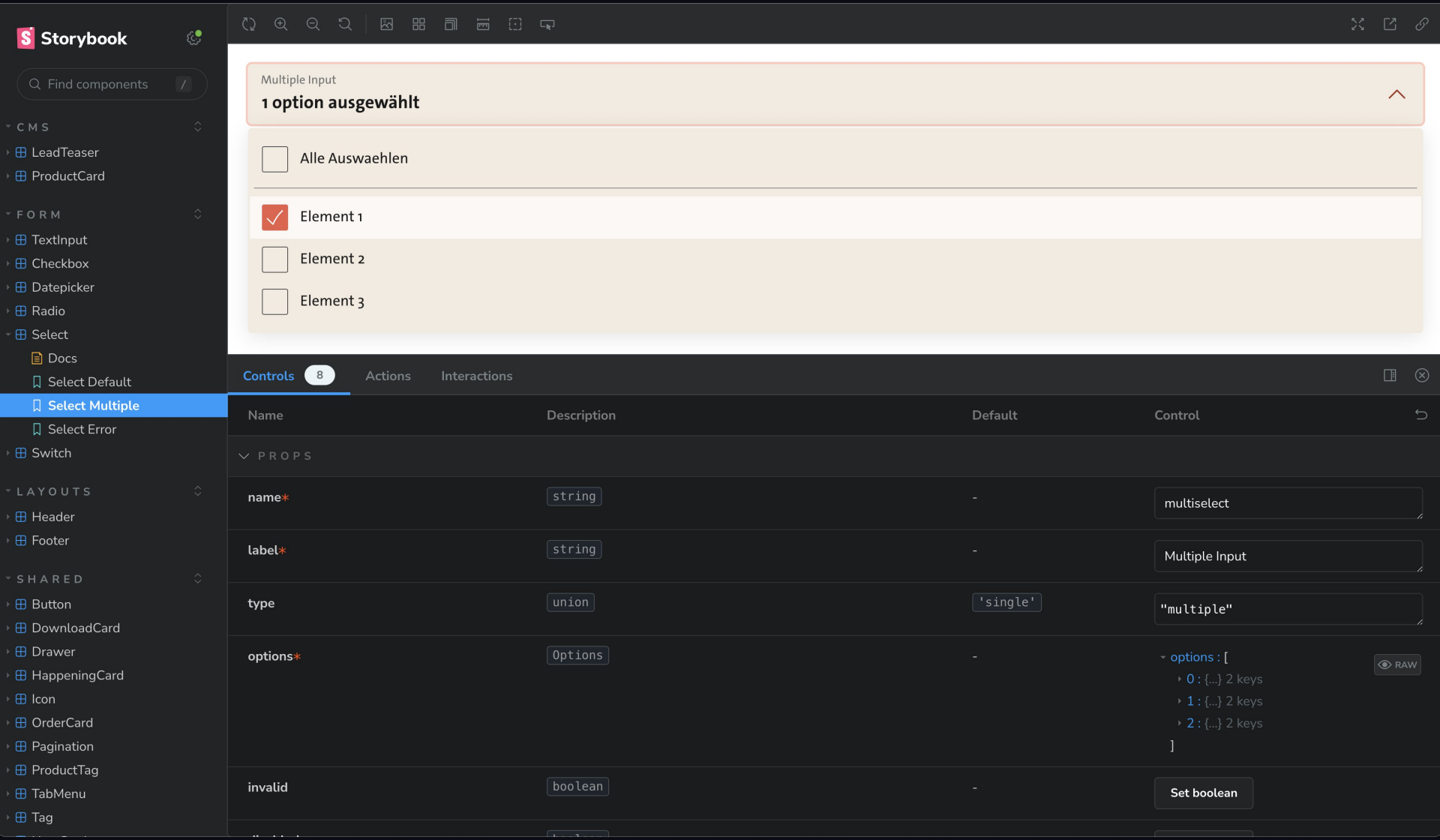
Integration in composable frontends project

```
/* start your composable frontends project */

npm run dev


/* enable nuxt dev tools */

npx nuxi devtools enable
```


Storybook
Find components
/
CMS
LeadTeaser
ProductCard
FORM
TextInput
Checkbox
Datepicker
Radio
Select
Docs
Select Default
Select Multiple
Select Error
Switch
LAYOUTS
Header
Footer
SHARED
Button
DownloadCard
Drawer
HappeningCard
Icon
OrderCard
Pagination
ProductTag
TabMenu
Tag
Multiple Input
1 option ausgewählt
Alle Auswaehlen
Element 1
Element 2
Element 3
Controls 8
Actions
Interactions
Name
Description
Default
Control
PROPS
name*
string
-
multiselect
label*
string
-
Multiple Input
type
union
'single'
"multiple"
options*
Options
-
options : [
0 : {...} 2 keys
1 : {...} 2 keys
2 : {...} 2 keys
]
RAW
invalid
boolean
-
Set boolean

Result

- isolated development
- independent components
- testable components
- documentation

sitegeist

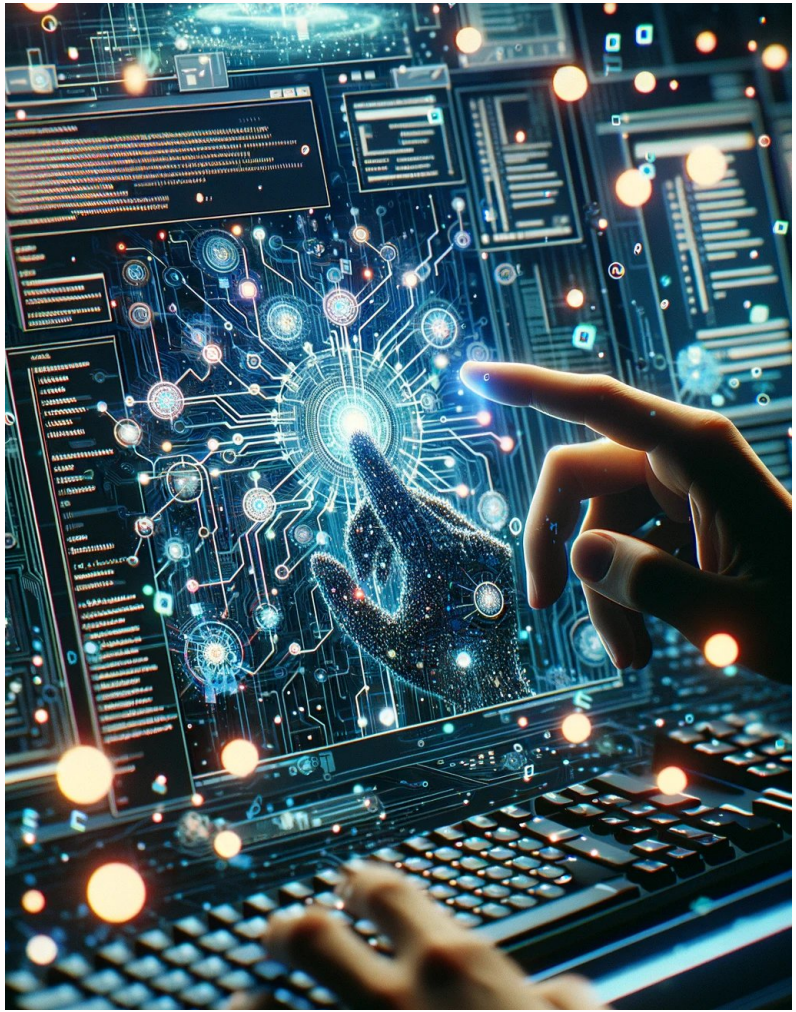
3. API client

Add own store-api endpoints

sitegeist

The new API client

- Dynamic generated typescript API client based on OpenAPI schema
- Every store or admin API endpoint with OpenAPI schema documentation will be available as types



Add OpenAPI documentation

```
/* add OpenAPI files to your plugin */

src/Resources/Schema/StoreApi/myStoreApiRoutes.json

src/Resources/Schema/AdminApi/myAdminApiRoutes.json
```

Add OpenAPI documentation

```
{
  "openapi": "3.0.0",
  "info": {
    "title": "SitegeistContract contract schemas",
    "version": "1.0.0"
  },
  "paths": {
    "/store-api/sitegeist-contract": {
      "post": {
        .....
        "operationId": "readContracts",
        "parameters": [],
        "requestBody": {
          "required": false,
          "content": {
            "application/json": {
              "schema": {
                "allOf": [
                  {
                    "$ref": "#/components/schemas/Criteria"
        .....
```

```
"responses": {
  "200": {
  "description": "Entity search result containing contracts",
  "content": {
    "application/json": {
      "schema": {
        "type": "object",
        "allOf": [
          {
            "properties": {
              "elements": {
                "type": "array",
                "items": {
                  "$ref": "#/components/schemas/SitegeistContract"
                }
              }
            },
            "type": "object"
          },
          {
            "$ref": "#/components/schemas/EntitySearchResult"
          .....
```

sitegeist

API generator: .env

```
/* create .env file */
OPENAPI_JSON_URL="https://SALESCHANNEL_DOMAIN"
OPENAPI_ACCESS_KEY="SALESCHANNEL_ACCESS_KEY"
SHOPWARE_ADMIN_USERNAME="SHOPWARE_CORE_USERNAME"
SHOPWARE_ADMIN_PASSWORD="SHOPWARE_CORE_PASSWORD"
```

API generator: load schemas

```
/* download schema for store api */
npx @shopware/api-gen@canary loadSchema \
  --apiType=store \
  --filename=storeApiSchema.json


/* download schema for admin api */
npx @shopware/api-gen@canary loadSchema \
  --apiType=admin \
  --filename=adminApiSchema.json
```

API generator: generate types

```
/* generate types for store api */
npx @shopware/api-gen@canary generate \
  --filename=storeApiSchema.json


/* generate types for admin api */
npx @shopware/api-gen@canary generate \
  --filename=adminApiSchema.json
```

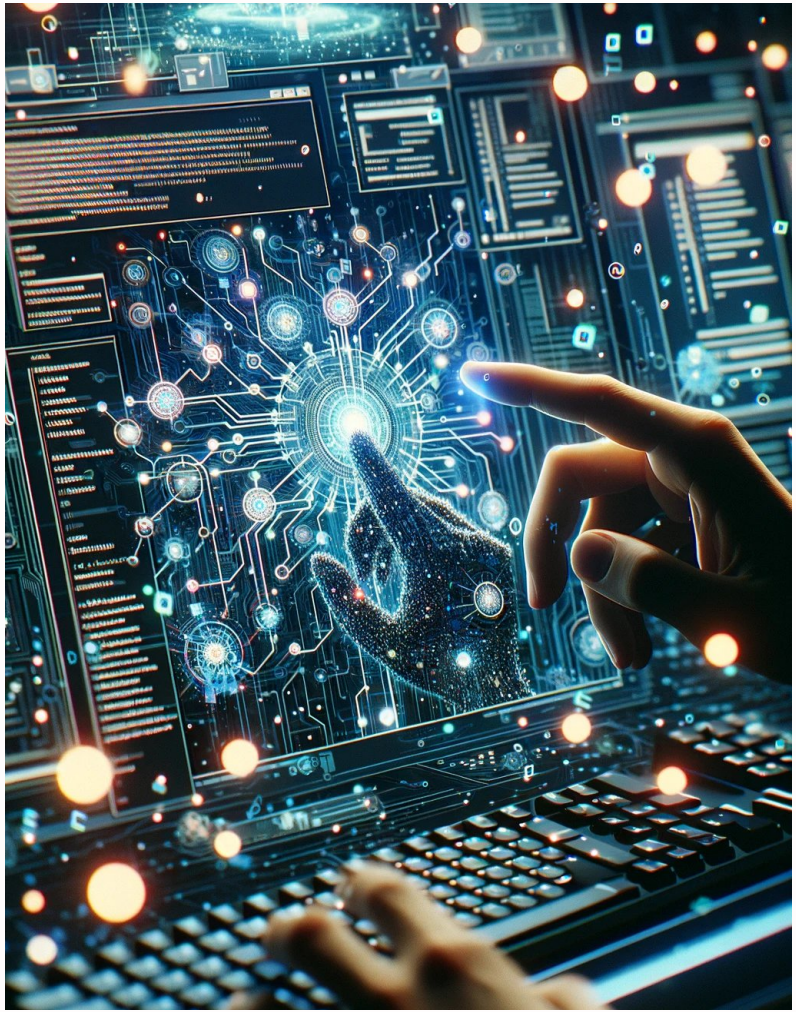
Import generated types

```
/* store api types */
import type { operationPaths, operations }
  from 'storeApiSchema';


/* admin api types */
import type { operationPaths, operations }
  from 'adminApiSchema';
```

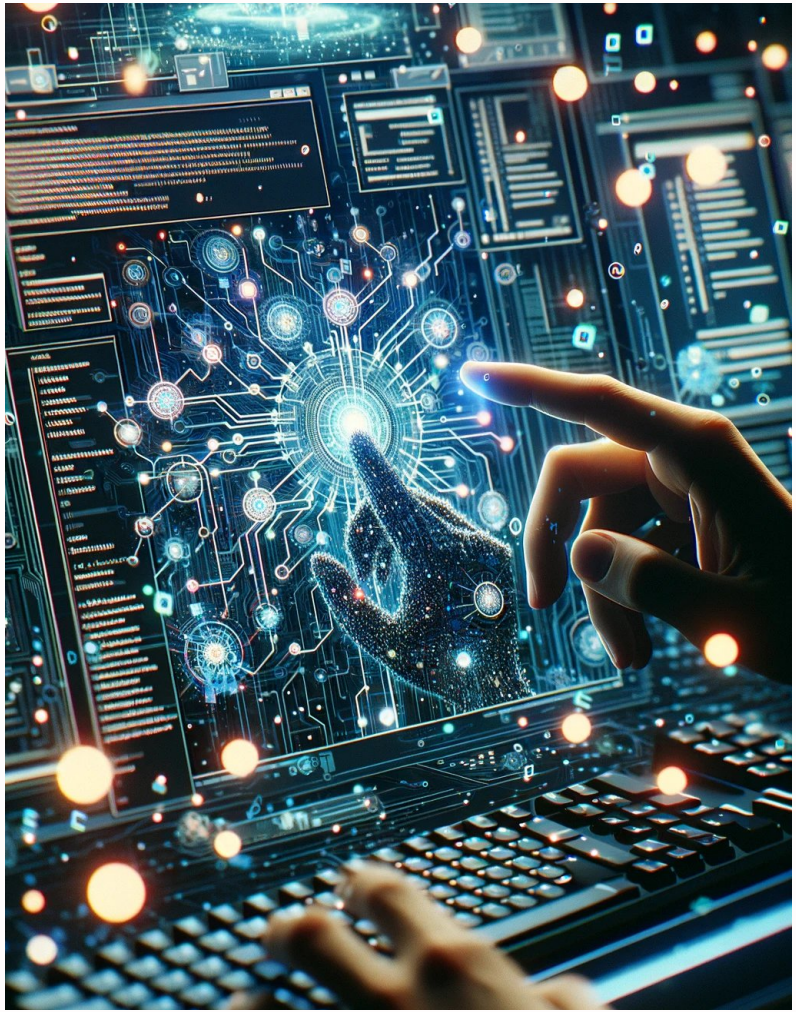
What else is possible?

- Connector for order management tool
- AWS lambda function with node.js
- Typescript objects instead of parsing JSON



Something else?

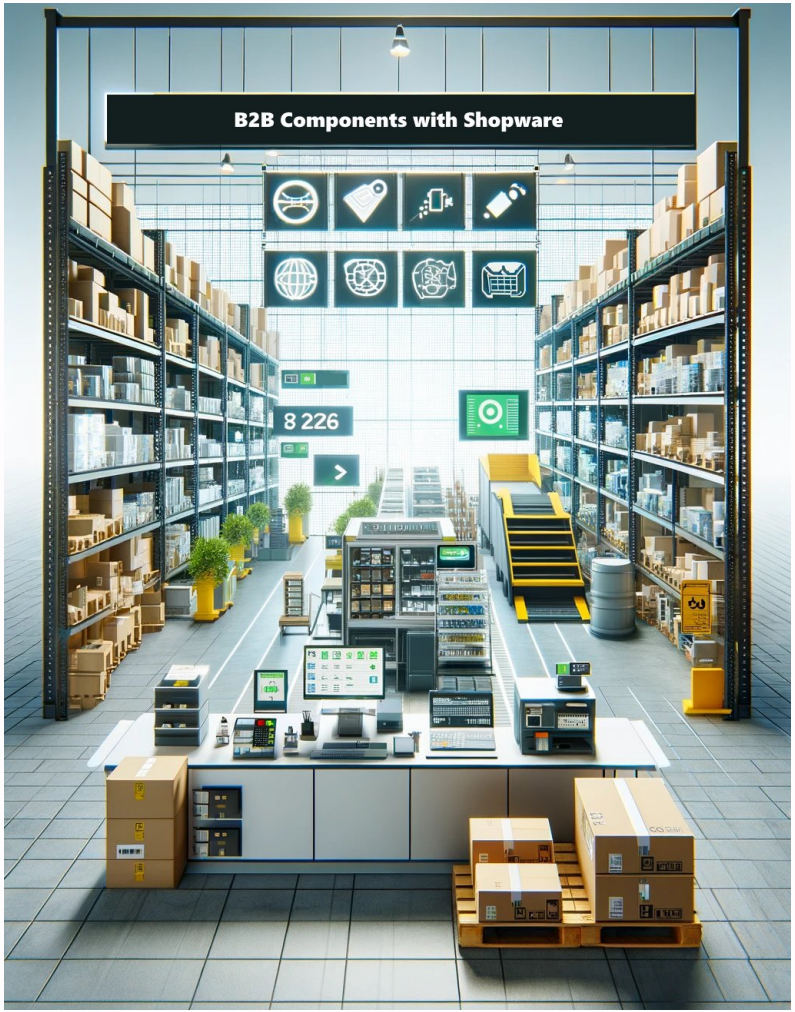
- In one of our next projects we will integrate Shopware to an existing Neos CMS website project
- We will use the composable frontends API client for integration



4. B2B components

Integrate B2B components to composable frontends project

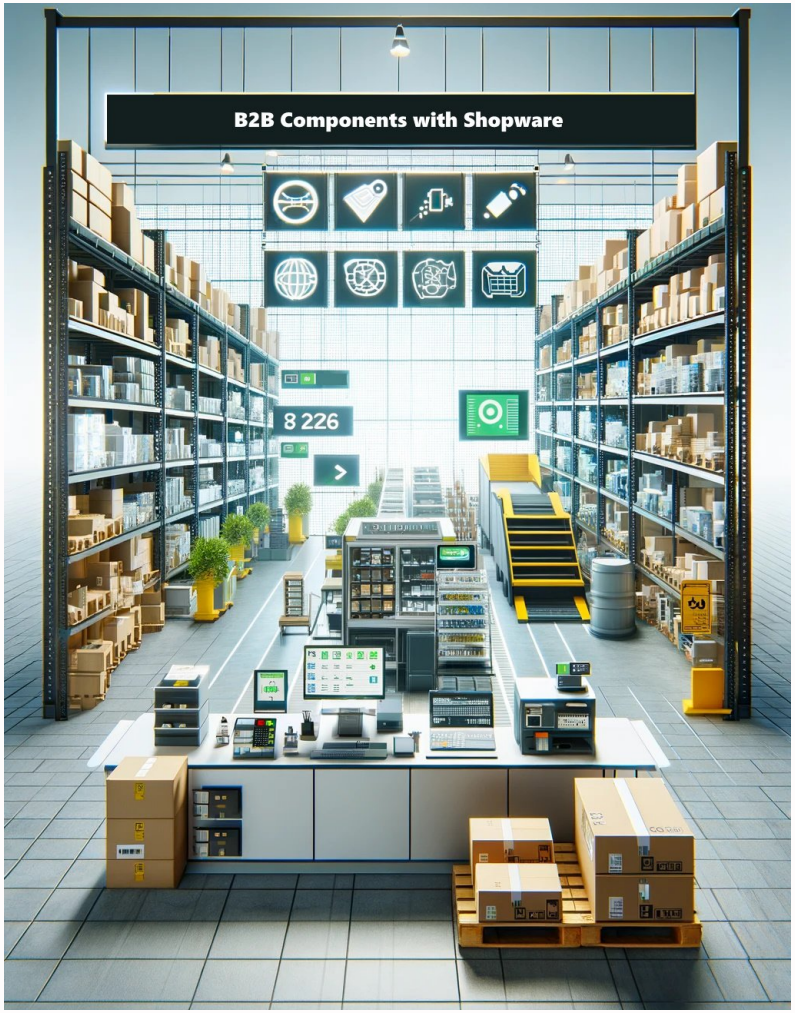
sitegeist



Employee management

- Store API available
- Types already part of the default types from the API client
- We just had to write composables for integration

sitegeist



Available modules

- B2B quote management is already part of the composable-next package and is ready to use

sitegeist

And composable for B2B module

```
export function useB2bEmployee(): UseB2bEmployeeReturn {
  const { apiClient } = useShopwareContext();

  const _storeB2bEmployees: Ref<Schemas['B2bEmployee'][]> = inject('swB2bEmployees',
ref([]));
  provide('swB2bEmployees', _storeB2bEmployees);

  async function loadB2bEmployees(parameters: Schemas['Criteria'] = {}):
Promise<void> {
    try {
      const result = await apiClient.invoke('readEmployees post /employee',
parameters);
      _storeB2bEmployees.value = result.elements ?? [];
    }
....
```

sitegeist

And composable for B2B module

```
async function createB2bEmployee(b2bEmployee: RequestParameters<'createEmployee'>):
Promise<Schemas['B2bEmployee']> {
  return await apiClient.invoke('createEmployee post /employee/create', b2bEmployee);
}

async function updateB2bEmployee(b2bEmployee: RequestParameters<'updateEmployee'>):
Promise<Schemas['B2bEmployee']> {
  return await apiClient.invoke('updateEmployee patch /employee/{id}', b2bEmployee);
}

async function deleteB2bEmployee(id: string): Promise<void> {
  await apiClient.invoke('deleteEmployee delete /employee/{id}', { id });
}
```

Conclusion

- Use storybook (or similar) for independent and testable components
- Think API first
- Write OpenAPI documentation

Thank you!

Any questions?

E-Mail: mueller@sitegeist.de

LinkedIn: https://www.linkedin.com/in/mmü

Landingpage: https://t.ly/cqAwq

sitegeist