FE Migration - 03 Build

Discussed at 28.03.

Notable changes so far:

- IE is not supported by Vue 3 & polyfill is not possible. Appropriate changes have been worked in.

Visual Decisions

We document & argue for the decisions taken which influence UX.

CSS

Options:

- A component framework like *Bootstrap* with premade components. While the existing components are rapidly integrated into the application and of high-quality, it is hard to create custom components from scratch.
- A semantic classes framework like *Tailwind* with utility css. All components have to be made from scratch, but the utility classes simplifies this process.

Approach:

- Use a component framework for standard components (like Modals). Building custom components is hard and expensive (lots of details, consistency, UX).
- Use Bootstrap 3 as it is already integrated & two bootstrap versions at the same time not possible.
- When building custom components, use a framework like tailwind.

Icons

Approach:

- Use fontawesome. Good vue.js integration. Massive library of icons (guaranteeing consistent styling), with tiny footprint (only includes used icons).
- Use Pro if possible. Icons look better (as subscription grants access to light styles).

Technical Decisions (Backend)

We document & argue for the decisions taken which only influence technical implementation in the backend.

API concept

In their extremes, two different concepts:

- Resource-based allows to reuse nodes, but has to cover all use-cases at the same time and authorization is more complex.
- Use-case-based fits some use-case perfectly, but leads to duplicated nodes.

Approach:

- Resource-based API as fewer nodes are easier to extend, test, maintain and migrate. Likely easier with future OAUTH migration.

API implementation

What should process the requests and send responses? Two different frameworks are realistic:

- Play framework as before. Needs some adaptions as resource-based requires more complex authentication. Provides few utilities, but battle tested. But little knowledge in team / effort could be wasted when someone has to pick it up.
- Spring framework as planned in the future. Reimplement already in correct framework, which eases integration. Provides many of the expected utilities out of the box. But unclear if integratable in current setup, little knowledge in team.

Approach:

- Prefer Spring to Play, as controllers have to rewritten anyways and therefore migration is for free. The better support for utilities (authorization, request parsing, ...) will likely even save time during migration.
- Test time-boxed whether spring framework is integratable. If integration fails, use play framework.

API integration

The API will ensure authentication (by HIN), authorization, validation and request deserialization. Then it executes the respective business functionality. In the end, it process the answer (either some response, or an error) by deserializing and constructing the HTTP response. How should it integrate with the business functionality?

- Service: Extract the business functionality from the existing controllers and place it in services. Minimal abstraction which needs to be cleanup later on.
- Facade: Define ideal functionality (POV controller) and let the facade handle the complex legacy integration.

Approach:

- Go with services for now, to avoid facades possibly slowing down the migration.
- For new functionality, implement it using the clean architecture approach.

Note: Concurrently, the backend is scheduled to be modularized. This however does not impact the API migration, as it starts at the center (domains), and will only touch the controllers later on.

Technical Decisions (Frontend)

We document & argue for the decisions taken which only influence technical implementation in the frontend.

Architecture

The structure of the translations can be freely specified, and there are no clear conventions to adhere to.

We aim that the architecture enables reusable, maintainable and easy to test components. This enables effective development and a consistent UI.

Approach:

- Separate different logic domains (layout, formatting, validation, web requests, interactivity).
- Keep domains together (to ease finding components).

Concrete approach (see example):

- `.` for the entry point of each app. It contains high-level layout (one column, to columns, ...) & connects to other parts of the page (like notifications).
- `./components` for components with a single purpose / no dependencies on same level. Prefixed with consuming layout, as width, height has to be known for effective design of low-level layout. Handles initial loading (incl. loading animations) before more specific components are loaded.
- `./components/Action` for components which execute some action (includes patch/post logic).
- `./components/Form` for forms (includes validation, autocomplete logic)
- `./components/Library` for application-agnostic elements. This includes form fields, modals, buttons, formatted datetime, loading indicators, ...
- `./components/View` for display of domain models, possibly in multiple variants suited for different contexts.
- `./localization` with the translation files & necessary aggregation logic (depending on translation architecture; see below).
- `./services` to abstract libraries, api, validation logic, ...

Example for the start page:

- `Index.vue` as the entry point defines the two column layout with the header. It references `components/TopBar.vue`, `components/LeftBar.vue` and `components/IndexDocuments.vue`. To the later, it passes the current folder.
- `components/IndexDocuments.vue` loads all documents belonging to the current folder while showing a loading indicator. When it is done, it passes the loaded documents to `components/View/DocumentsTable.vue`
- `components/View/DocumentsTable.vue` references `components/Action/DeleteDocumentsButton.vue`. It displays the documents, using reusable components such as `components/Library/HumanReadableDateTime.vue`. It manages which documents are selected.
- `components/Action/DeleteDocumentsButton.vue` wraps its content into `components/Library/ModalWithConfirmButton.vue`. Its content is the `components/Form/DocumentsConfirmDelete.vue` component, which displays the to-be-removed documents and requires the used to check a checkbox. When the `ModalWithConfirmButton` emits the `confirmed` event, it sends the DELETE request to the server.

The library likely includes Modals, Buttons, Dropzone, PaginatedTable, ...

Global state

When state has to be accessible in many components (either from parent into a children deep down, or from child into another child) it leads to complexity in all components "on the way": They have to listen to change state events, process them & possibly raise more events.

A store like pinia can help. Instead of raising events to the parent, the component updates the state in the store. Stores can easily be injected into as many components as needed. To avoid a mess, stores implementations like `vuex` introduce an additional abstraction layer: Instead of directly modifying state, a function has to be called which the modifies the state. Such a central store needs to be carefully designed to avoid huge stores.

As an alternative (for the parent injects into a child deep down issue), provide / inject can be used. This pattern essentially hides the properties passed through intermediate components.

Approach:

- Wait with integration of stores or provide/inject until clear need arises.

Single-File components

There are two fundamental way of how to structure vue application:

- Options API is in "object format"; functionality is placed in appropriate places of a javascript object (for example `props` for properties, `methods` for methods, ...). Functionality is separated by technical concerns; as components have to be small anyways, this is however not a big issue.
- Composition API is more script-like; different technical building blocks of the same concern can be grouped together. The technical separations (for example reactive properties) have to however be explicitly initialized, requiring more vue.js knowledge.

Approach:

- Use the Composition API. More natural to write with TypeScript, and good results with the new document view. The additional effort in learning vue.js fundamentals is acceptable.
- In .vue files, only `style scoped` is used. Global styles should be placed in appropriate global `.css` file (to make the global classes visible to the developers).

Routing

The routing should be done by vue.js in the last part of the rewrite effort. As routers cannot pass state (except the route URL), this requires a central store for state that should be shared between pages.

Approach:

- Do routing refactoring after the main migration finished.

Validation business logic

Validation has to be implemented again by the API. To avoid code duplication, complex business logic should be avoided in the frontend, rather the API should return an appropriate error code.

Translations

Vue I18n suits our requirements. Easy, well-maintained and good experiences in the past.

Translations are specified as a JavaScript objects. Easy to implement are therefore translations strings from .json files.

Structure of translations

The structure of the translations can be freely specified, and there are no clear conventions to adhere to.

We aim that every rule of the translations structure fulfills the following conditions:

- Simple to remember (to avoid wrong placements).
- Avoid too large files (to ease finding strings; simplify work of translators).
- Only allow a single place where a translation string can be possibly located (to avoid repetition).

Approach:

- *Separate file per language:* Each language gets their own file.
- *Mirror architecture:* The translation strings are where the object is located in the filesystem. For example, all translations used in `components/View/Modal.vue` are located in `view.modal.*`
- *Unify domain*: The business domain is only represented once. For example, all translations related to the patient (like first_name, family_name) are located in `domain.patient.*`
- *(optional; only if scaling not yet good enough) Separate file per top-level architecture:* For each top-level architectural concern, add separate language file.

Import from Scala

Scala already has translations specified in `messages` files. It is possible to convert these into the .json files required by Vue i18n.

However, the strings still require heavy restructuring (to adhere to the structure defined above). Further, as parts of the UI change, so should also the translation strings.

Approach:

- Manually take over scala translations which still match. However, most labels likely still need a re-translation.
- Possibly write a migration script for other languages.
- Possibly streamline the process of translation to save PO time / increase re-translation iteration.

Building

We use Vite, as recommended by Vue.js.

The build must notably include:

- `eslint` for style checking and automatic fixes in Vue.
- `stylelint` for style checking and automatic fixes in `sass`, `scss` files.
- `postcss` with autoprefixer for improved browser support
- `babel` configured to use `corejs` for improved browser support (using browserslist)

Further, to ease development, the build includes an opt-in pre-commit hook which executes fixers of linters.

The build should be strictly CSP compliant.

Testing

We use Vite as a build tool, therefore recommended by Vue.js is:

- Vitest for unit and components tests. Vitest is rather new, and the more established Jest is also supported. Migrating between different framework should however not be too difficult, a the fundamental concepts remain the same.
- Cypress for new E2E tests. Cypress is not selenium based, rather directly injects JavaScript into the browser. It promises less flakyness.

Current BlueConnect testing:

- Many Selenium tests written (ca 200), use-case based
- Smoke tests (ca. 20 tests) which test the most important use cases.
- Adapting Selenium tests to new UI should be reasonably simple.
- Testing is open to try out Cypress.

Approach:

- in `components`, focus on black-box component tests. No complex business logic in frontend, hence whitebox / unit testing likely not that useful.
- in `services`, focus (obviously) on unit tests.
- E2E tests strictly only for behaviors which cannot be tested in component tests.
- Selenium use-case based E2E tests are adapted, using `data-test-id` selectors.
- For each page, a new cypress E2E test is developed, focusing on the technical interactions between components.

Dependency strategy

Avoid dependencies to reduce maintenance effort. Favor inclusion of source directly into project, or avoid functionality entirely.

Examples:

- `BootstrapVue` provides vue components for bootrap components. Abstraction helps with initialization of JS features, or setting correct CSS classes. However, this is almost no advantage (as trivial to implement self), and the drawbacks are severe: Unstable, have to use docs of BootstrapVue (worse than bootstrap docs), lags behind (still stuck at bootstrap 4, although 5 is out for quite some time). *Use bootstrap directly instead.*
- `atom-spinners` provides spinners. However, only single spinner needed (not all 10), vue adapter broke from vue 2 vue 3. *Include file with the one spinner actually needed directly into the project sources.*
- `vuealidate` provides validation features. However, easy to implement self (rather than understanding the library); did not properly upgrade to Vue.js 3.
- DataTables like ag-Grid require lots of knowledge how to interact with the library, while probably being overkill for our use case. Front-end pagination, filtering and sorting is simple to implement in vue.js; although needs to be carefully architectured to not overload the component.