Contents

Developer Guide

Introduction

Aim of this guide is to help you understand how we work and what tools we use. It will aid you in setting your computer up, so that you can easily jump into current workflow. Also, this guide briefly covers standards and practices that we apply in our daily work. We hope that this guide will answer all your questions about how we develop applications here in SwingDev. If you have any questions, don't hesitate to ask us (you can find all needed contacts in Who you can ask [page 9] section). Welcome aboard!

About the Company

Working at SwingDev means that you will be surrounded by people who are pushing the limits of software product development.

You will have the opportunity to give input on your tasks and you will have an actual impact on the products you build. On a daily basis, you will work in close collaboration with a team of professionals — IT specialists, project managers, designers, and other creatives who are always looking for ways to learn from each other.

At SwingDev, our mission is to make software product development and consulting an efficient, approachable service done with integrity.

We need your skills and dedication to make it happen.

Environment setup

Architecture

Coding practices

Writing code using widely recognised and opinionated practices makes our job easier. In our work, we use following rules:

- SOLID
    - **Single responsibility principle -** a class should have only a single responsibility (i.e. changes to only one part of the

    software's specification should be able to affect the specification of the class).

    - **Open/closed principle -** ”software entities ... should be open for extension, but closed for modification.”
    - **Liskov substitution principle -** ”objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.” See also [https://en.wikipedia.org/wiki/Design_by_contract].
    - **Interface segregation principle -** ”many client-specific interfaces are better than one general-purpose interface.”
    - **Dependency inversion principle -** one should ”depend upon abstractions, [not] concretions.”

- KISS - keep it simple!
- DRY - don't repeat yourself
- YAGNI - you ain't gonna need it
- Finite State Machines - can be helpful [page 29]

Application structure

To make our application scale well, we divide it into several layers — UI, logic and data. Also, a well-thought project structure helps with this task.

For more details, see:

- Project structure [page 32]
- Keep your UI, logic and data layers separate. [page 34]

- Abstraction layers - not too little, not too many. [page 37]

Error handling

We favour proper, well described error handling. It's not only a benefit for the user, but also for the developer. We recommend creating an exception hierarchy with custom error types.

For more details, see:

- Exception hierarchy [page 39]

Code

Languages

The primary language in the browser environment is JavasScript. Its newest edition, ES6, adds a bunch of useful features. Currently it's the golden standard in web development and we recommend to use it.

ES6 is great but has one major flaw — no type checking. At the moment, there are two solutions for this issue — [https://flow.org/en/] and [https://www.typescriptlang.org/]. We prefer the latter in our daily work.

For more details, see:

- JavaScript (ES6) [page 47]
- TypeScript [page 49]

Frameworks

Frameworks are a must in modern web development. They include a number of patterns and helpers which improve application development process.

One of the most popular frameworks is React. This library is a component focused rendering engine, it provides good maintainability and community support. Moreover, it gives high-performance, thanks to Virtual DOM.

For more details, see:

- React [page 42]
- Vue.js - *work in progress...*

Markup and Styles

Even if you prefer using a front-end framework over plain HTML, you still need to know basic rules of writing markup code. With HTML5 introduction, we change our way of writing HTML code, discarding divs in favour of semantic elements.

CSS seems to be a simple language, but due to it's cascading and global nature you can quickly fall into a pitfall. Consider using preprocessor such as SCSS and any class naming convention.

For more details, see:

- Markup (HTML) [page 45]
- Styles (SCSS) [page 51]

Linter rules

You should always lint your code! We do it, and you should do it also. It makes your life easier.

For more details, see:

- [https://swingdev.quip.com/pUiGAe3wrz7a]
- [https://swingdev.quip.com/MwWaAavZvLz4]
- [https://swingdev.quip.com/b0LwAaC9rQmO]

Application behavior

User experience

As front-end developers, we are responsible for delivering excellent user experience. We need to keep in mind that all internet users are different from each other, and most importantly — from you. Some of them will be on a poor cellular connection. Some of them will have disabilities.

For more details, see:

- Real-life connections [page 53]
- Accessibility [page 57]

Performance

Poor connection may not be an only issue. Web can be accessed from a whole spectrum of devices — from old to modern ones. You have to care about your application's performance more than you think.

For more details, see:

- Battery performance matters [page 58]
- Loading performance [page 60]
- Optimize Images and Videos [page 65]
- Animations performance [page 68]

Data handling

Proper data handling can significantly improve user's experience. You should always propagate latest data to your views and deal with data conflicts appropriately.

For more details, see:

- Fresh data should propagate across the whole UI [page 73]
- Handle conflicts consciously, not accidentally [page 76]

Data fetching

Usually, front-end applications fetch data from back-end APIs. Poor request handling can cause extended response time or even break your back-end. You should incorporate lazy-loading and batch loading into your data fetching services.

For more details, see:

- Lazy-load in the UI [page 81]
- Batch loading [page 83]

Tests

- Write as many tests as you need to be confident in your code.
    - Prefer end-to-end over integration tests for testing UI flow.
    - Prefer unit tests for small pieces of logic.
    - Don’t write tests that will never fail.
    - Test behavior, not implementation.
    - Don’t hesitate to remove tests which give none to little value.
- Keep end-to-end tests in a separate repository, separate docker file.
- Run unit / integration tests within Docker — that’s the way they will be run.

Who to contact

Marcin Mincer (COO) - marcin@swingdev.io

Tomek Kopczuk (CTO) - tomek@swingdev.io

Kacper Kula (Developer Evangelist) - kacper@swingdev.io

Mirek Ciastek (Senior Front-end Developer) - mirek@swingdev.io

Computer setup

Background

The computer is our primary tool. It seems obvious, but we tend to forget about a proper setup. The rightly configured computer can save you much time during development. It's better to do it once and right than to fight with your machine.

Rules

Before you start working check if you

- Installed [https://git-scm.com/downloads]
- Installed [https://brew.sh/]
- Installed [https://nodejs.org/en/] with NPM (LTS is recommended)
- Installed [https://github.com/creationix/nvm] (Node Version Manager)*
- Installed [https://yarnpkg.com/en/]*
- Installed [https://docs.docker.com/install/] from official repository (Homebrew version is broken and won't work)

* *these we recommend to install by Homebrew*

Choose any editor you want (e.g. WebStorm, Visual Studio Code, Sublime Text), but be sure that you installed plugins for:

- ES6 / React / Vue.js

- Typescript
- Eslint / Tslint
- Stylelint

Project setup

If you are setting up the project, remember that it should be easily maintainable for other developers, devops and should fit into our deployment setup.

You can experiment with architecture, technologies and solutions but within safe boundaries which help us speed up the process.

All our new projects use bootstrap repository which is a starting point for the project. Bootstrap repository contains docker-compose which describes all modules / services. All additional configuration like nginx configuration should be there as well. For a more detailed information see our [https://github.com/SwingDev/bootstrap-bootstrap] official repository. Every new project should use this tool.

Also, please use other application bootstraps as an example, to solve more complex problems, and to speed up the process. All the bootstraps we have within the company came from months of work in other projects so almost all the decisions made there are battle-tested.

Do not hesitate to question the solutions there, but give them a chance.

If you don't know where to look for a bootstrap, contact other developers working in this technology within the company - they will show you the way.

- Your project will be developed, run, tested and built through Docker.
  - Develop locally in Docker [page 27] - it's enabled for live-reload.
  - Run tests in Docker [page 26] environment.

    - Have the required commands [page 25] working at all times in Docker container.
    - Handle per-environment constants [page 28] according to our rules.
- Projects must be runnable in a complete state through `docker-compose`, using a `<project name>-bootstrap` repository.
    - All third party services need to have their own shared local burner environment [page 15] set up.
- Be prepared for asset caching [page 23].

Bootstrap repository

Rules

- Start from: https://github.com/SwingDev/bootstrap-bootstrap
- Add your modules.
- Add a docker-compose file for all your services.
- Develop using this repository, and keep it working at all times.
- Enjoy!

Local burner account for all 3rd party services

Background

We believe in working with real services, not mocking them. The sooner a potential problem is found in the software lifecycle - the better.

Hence - we require all third party services to be configured and working using their own burner environment on a local Docker run from the bootstrap repository - with no additional config necessary.

This way you don't need to mock 3rd party services, or pretend you tested if it works - you know it does.

Rules

- All third party services (Google Analytics, authentication - like Firebase / Auth0, etc.) need to have their own environment set up for local development, shared between all developers.
- This environment must be treated as 'absolutely insecure'.
    - **No** private, confidential or in any way real data can be used while developing locally.

How to test this behavior?

When I run the app from it's bootstrap repository:

- Can I use every single feature of the application?
    - E.g. can I send and receive emails if such functionality exists?

Implementation hints

Look at e.g. [https://github.com/SwingDev/swg-certalerts-bootstrap/blob/master/docker-compose-common.yml] and how it's configured for Mandrill and Auth0.

Git

Background

Git is the tool we use everyday. For many this might sound obvious, but it is still worth mentioning. Try to use git whenever possible, even if you are bootstrapping very small project with just few lines of code - this is really benefitial.

Rules

- Commit Often
    - Git is a great tool if you commit small changes often. It helps both with rollbacks, identifying potential source of error and helps the reviewer follow your train of thought.
- Use git-flow
    - Whenever possible, use git-flow for your projects.
- Make useful commit messages
    - It is easy, when in rush, to commit the code with the message like “Fix”, but this ruins the whole idea of having history of the commits. Imagine that in the near future you might need to decide if this commit is a part of a bigger story and should be merged to the main branch or not - is this message helpful?
- Don’t Panic

- Some parts might be overwhelming. It seems simple at the surface but sometimes it might backfire. Remember, as long as you pushed your code to the remote server, your changes are safe.

How to commit your code?

When you are finished with current work it's worth to commit it. You should commit a separate, working piece of code, which can be easily reverted without any harm. That's why it's good to make your commit atomic. Below are good articles about atomic commits:

- https://www.freshconsulting.com/atomic-commits/
- http://adopteungit.fr/en/methodology/2017/04/26/how-to-do-atomic-commits.html

After you staged your changes it's worth to add meaningful commit message, so that other developers will know what you did. Here are some tips how you can do it:

- https://chris.beams.io/posts/git-commit/
- https://robots.thoughtbot.com/5-useful-tips-for-a-better-commit-message

Below is a git commit template that we use in our company. Setup it for you local environment:

```
# If applied, this commit will: ⊠      ...limit ⊠   ...hard
   limit ⊠

Reason:
```

```
Relates to:

# RULES:
# 1. Separate subject from body with a blank line
# 2. Limit the subject line to 50 characters (69 is a hard
    limit)
# 3. Capitalize the subject line
# 4. Do not end the subject line with a period
# 5. Use the imperative mood in the subject line
# 6. Wrap the body at 72 characters
# 7. Use the body to explain what and why vs. how
```

See how you can add this commit template to your git setup [https://robots.thoughtbot.com/better-commit-messages-with-a-gitmessage-template].

How to Create Pull Request

Pull Request tells other developers what changes you want to include into core codebase. Making meaningful Pull Request descriptions helps your teammates understand what changes you want to introduce. Here are some tips how to write good Pull Requests. If your project have a standard pull request format, stick with it! Below you can find useful materials:

- https://blog.github.com/2015-01-21-how-to-write-the-perfect-pull-request/
- http://blog.ploeh.dk/2015/01/15/10-tips-for-better-pull-requests/
- https://readwrite.com/2014/07/02/github-pull-request-etiquette/

Code Review

Always do code review. If you are a single developer working on a project, find other developer who can make you quick code review. It is essential to have someone reviewing your code for several reasons. Firstly, it is easier to someone to criticise your solutions and find some pitfalls you might fall into. Also, it forces you to write clear, readable and consistent code.

Sign Your Git Commits

GitHub provides an easy way to sign your commits. It adds additional security level. To do so, first generate your key and setup it on your local machine:

```
gpg --gen-key
```

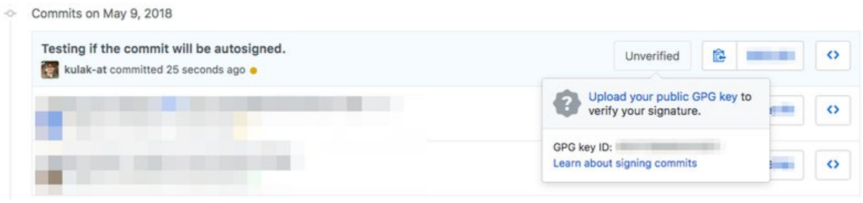
```
git config --global user.signingkey 0A46826A
```

If you want to autosign all the commits for your current project, use the following command in the project directory:

```
git config commit.gpgsign true
```

If you want to setup autosign globally and make all commits you are making signed by the GPG key, add --global flag.

Then you can setup the key on your GitHub account:


Commits on May 9, 2018

Testing if the commit will be autosigned.

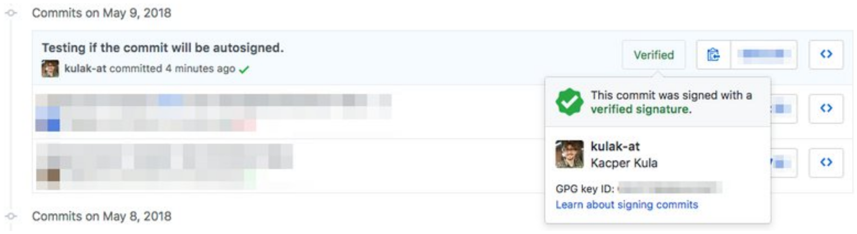
kulak-at committed 25 seconds ago

Upload your public GPG key to verify your signature.

GPG key ID:

All you need to do now is to add your public GPG key:

```
gpg —armor —export KEYID
```


Commits on May 9, 2018

Testing if the commit will be autosigned.

kulak-at committed 4 minutes ago

This commit was signed with a verified signature.

kulak-at
Kacper Kula

GPG key ID:

Commits on May 8, 2018

Continuous Delivery

Background

We use Continuous Integration and Delivery practices.

Continuous Integration ensures that small code part is delivered to repository frequently and integrated with existing codebase without any errors (more you can read on [https://en.wikipedia.org/wiki/Continuous_integration]).

Continuous Delivery practice guarantees that software can be reliably released at any time (more informations you can find in [https://en.wikipedia.org/wiki/Continuous_delivery]).

For CI / CD we use Circle CI.

Prepare for asset caching

Background

Remember that all your files will be cached by - the browser, a proxy, CDN - you never know what, when and for how long.

It's necessary to develop under this assumption from the very beginning.

Rules

- Version **all** of your files - e.g. by having Webpack append a hash to bundle file names
- Have a list of files that need to be invalidated on deploy (if any other than `index.html`) listed in the README.
    - Otherwise the assumption is - nothing is to be invalidated apart from index.html.

How to test this behavior?

Build the production version and look for fetched files under the Network tab of your Dev Tools.

Any filename that isn't unique will cause problems and will be cached.

Implementation hints

Use content hash in your filenames.

Required yarn commands

Background

These commands will be used by different parts of the whole process - from CI, through actual servers running your app. Therefore it is vital they are being kept in 'always working' state.

Developing strictly in Docker helps to achieve this goal with next to no additional work, but as 'build' command triggers the 'production' pipeline - it's important to test it as well after making any changes to the pipeline (webpack etc.).

Rules

- Always keep these commands working:
  - `yarn start` - to fire up a local web server with hot-reload and all the works on port 8080
  - `yarn build` - to build a production bundle to `/app/dist`
  - `yarn test` - to fire up unit / integration tests
- Test `yarn  build` and built assets every time you modify the Production WebPack pipeline.

Run tests in Docker

Background

This is the way that CI and your colleagues will run your tests. Don’t try to be different ;-)

Rules

- Command to run tests:
    - `docker-compose run –rm <your service name> yarn run test`

Develop in Docker

Background

We recommend using Docker for working on your application. Docker is a platform which allows you to run your application in specific virtual environments. This ensures that your application will work no matter what OS or software you currently use. More about Docker you can find here.

To get familiar with our Docker setup, you can take a look at the example setup.

Rules

- if something does not work / would require you to go around Docker - fix it or tell us, and we'll fix it together.
- this way we can be sure nothing changes on CI and after deployment > you're looking at the same application

Per-environment constants

Background

It’s often necessary to bake some environment-specific information (api keys, urls, etc.) into our frontend builds. This is how we do it.

Rules

- enable WebPack / Gulp to use `.env` file as their source
    - CI will build the right one for the environment as a building step
    - Locally you’ll have `.env.local` bind-mounted to `.env`

Finite State Machines

Background

https://en.wikipedia.org/wiki/Finite-state_machine

We utilize them to keep state of the application consistent and deterministic.

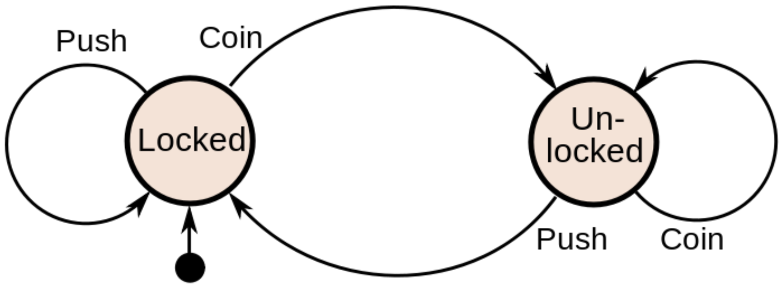
By limiting what actions do and when we prevent side effects of asynchronous actions being completed at unexpected times.

They also simplify complex step-by-step or multi-variable logic by dividing it into small independent pieces orchestrated by other small independent pieces.

For complex flows - use FSM that uses state of other FSMs - to keep complex flows simple.

Same principles apply to UI components behavior orchestration, just on a micro UI level.

Architecture



- State is the single source of truth for 'where the application / module / view / logic' is now.
- Conditions that make the state transition from X to Y are well defined. If they happen > state transition is made, and the hooks are fired.
- No other condition cause the state to change.
- FSMs can watch other FMS's state, and use it as their own conditions.

Typical use cases:

Usual leaf FSM graph

- started > processing
- processing > success

- processing > failed (retry logic to go back to processing) > failure

Usual non-leaf FSM graph

- started > fetching_data (combines a few leaf FSMs)
- fetching_data > waiting_for_subFSMs_success
- waiting_for_subFSMs_success > success
- <any state> > failure
- retry logic can also be handled on this level, esp. if you need to retry the whole flow, or do rollbacks

Usual orchestrating app logic top-level graph:

- started > step1
- step1 > step2
- step2 > step1
- step2 > step3
- step3 > step2
- step3 > success
- note that failure is usually not handled at this level, as it should have been handled by sub-FSMs on their level (e.g. single-step level), or just by exiting this particular flow

Project structure

Background

Good project structure helps you and your teammates to find quickly part of the application. Moreover, well-thought project structure increases your application's scalability. It leads to less painful refactoring.

Rules

- Divide your application into logical parts
    - For example move views components to *views* directory
- Place files according to their scope
    - It's good to have common helpers directory, but when helper is used only in one place, keep it near.
- Always keep third party libraries in *node_modules,* unless it's necessary
    - When you need to copy library to your project, add proper note to README file
- Add README - it really makes on-boarding easier
- Divide dependencies to *dependencies* and *devDependencies* in *package.json*
- Divide styles to global and component scoped
- Place folder with component's tests in it's directory

- Organise *images* directory according to one of the conventions:
    - by their file type
    - by their function

Implementation hints

The example of good project structure you can find in our *react-ts-boilerplate* [https://github.com/SwingDev/react-ts-boilerplate/tree/master/src].

UI / logic / data layers

Background

Good layer separation allows for:

- Reuse of components
- Single-file, single-place changes of: ui, logic, behavior and api
- No regressions

Rules

UI layer - Dumb Components

Do:

- take data, produce UI
- take input, produce actions
- orchestrate micro-level behavior in the UI realm (e.g. an input field logic)
- initiate data layer / services layer data fetch if data unavailable
- handle loading state of the data layer / loading state of the data action
- handle failure state of the data layer / failure of data action

Do not do anything else, for instance:

- Fetch data
- Orchestrate application logic / flow

Logic layer - Smart components / Services / Actions / Controllers

Do:

- Use data layer to save / populate business domain objects
- Orchestrate application logic / flow

Do not:

- operate on DTOs

Data layer

Read about:

- [https://en.wikipedia.org/wiki/Data_access_object]
- [https://en.wikipedia.org/wiki/Data_transfer_object]

Data layer does:

- Fetch DTOs
- Transform DTOs > business domain objects

Usual data flow:

- Logic layer demands access to certain data

    - keeps it up to date with used cache changes
- Data layer checks cache
    - does nothing if populated
    - deletes and refetched when stale
    - refetches when none
- UI layer takes data from global application state / logic layer state

Abstraction layers

Background

Having too little abstraction layers cause the whole code to be one-off, not reusable, and not able to adhere to the ’single responsibility rule’.

Having too many causes unnecessary code to be written - at the same time this code is usually boilerplate, and quick to write - making this a much smaller problem than having too little layers.

Rules

- Keep your UI, logic and data layers separate. [page 34]
- Favor abstractions:
    - where components tend to evolve quickly during project ramp-up, e.g:
        - HTTP fetching layer (fetch through own abstraction, rather than straight through axios / etc.) - unless it’s already abstracted away (per Angular > 2) - for global control of automatic retries, exceptions abstraction, etc.
        - Authentication logic - as it tends to evolve a lot with sudden ACLs / changed behavior / added opt-ins / popups, etc.
    - where components might be replaced (API calls)

    - where logic seems a bit too complex
- Always write interfaces, and make sure to never rely on private / protected methods and properties.

How to test

- Think of your code from a perspective of someone who has no idea about the codebase and needs to:
    - add a feature
        - will he be able to do that without knowing current implementation details?
    - change behavior of a feature
        - will he be able to do that in one place
            - is it easily findable?
        - are there any quirks that need to be known?
    - change the way e.g. data is being returned from the API (e.g. from one endpoint to having to hit 2 endpoints and combine the results)
        - does it require a change in any components that use this data?
        - is it easily implementable in one place

Exception Hierarchy

Background

Exceptions are good - they simplify error handling and increase readability.

Rules

- Be aware of your exception hierarchy, and how it changes across the application layers.
- Catch exceptions early, and transform them from very specific low level exceptions to more generic, high level ones.

Implementation hints

Have base exceptions you’ll use all over the application

- ’temporary problem’
- ’permanent problem’
    - ’uncorrectable failure’
    - ’critical internal failure’

Create exceptions inheriting from these base exceptions to constrain possible exceptions being thrown from services:

- example
    - URLFetchingService with GET method
- exception hierarchy
    - BaseTemporaryErrorException
        - URLFetchingRemoteNotAvailableException - for 5xx and connection problems
    - BasePermanentErrorException
        - BaseCriticalInternalFailureException
            - URLFetchingCriticalInternalFailureException - for any situations that could indicate that something's very wrong (eg. can't serialize passed DTO to JSON) - so that we as developers are informed about this
        - URLFetchingNotFoundException - for 404
        - URLFetchingBadURIException - for internal axios exceptions dealing with uri
- now an example service that enables user to fetch remote urls would look like:

```
try {
    return URLFetchingService.get(uri, params);
} catch (e) {
    if (e instanceof BaseTemporaryErrorException) {
        return retry();
    } else if (e instanceof URLFetchingBadURIException) {
        this.addErrors({uri: 'Wrong URI format.'});
    } else if (e instanceof URLFetchingNotFoundException) {
        this.addErrors({nonfield: 'Resource is long gone.'});
    } else if (e instanceof BasePermanentErrorException) {
        this.onFailure();

        // this will:
        //  - fire up a notification
        //  - notify us if e
    is BaseCriticalInternalFailureException
        ErrorService.dealWith(e);
    } else {
        throw e;
    }
}
```

React

Background

> React makes it painless to create interactive UIs. Design simple views for each state in your application, and React will efficiently update and render just the right components when your data changes.

We choose React as our main front-end framework. It’s one of the most popular frameworks nowadays. It’s fast and has pretty low learning curve. If you don’t know React, this course can give you good overview. React is not a 100% percent MVC-like framework. It only allows you to render dynamic views using JavaScript. That’s why developer needs to add all required stuff by himself.

For managing state in React we use Redux or Mobx. Below are good courses for both of them:

- Redux - https://egghead.io/courses/getting-started-with-redux
- Mobx - https://egghead.io/courses/manage-complex-state-in-react-apps-with-mobx

Depending which state manager was used, we use different libraries to manage side effects:

- Redux - Redux Thunk or Redux Saga
- Mobx - side effects are handled by state manager itself

Rules

- Use [https://hackernoon.com/react-stateless-functional-components-nine-wins-you-might-have-overlooked-997b0d933dbc], when you component doesn't need to keep state or event handlers
- Always keep application state outside from component's state
- Name your event handlers properly, using naming convention ([https://jaketrent.com/post/naming-event-handlers-react/])
- Always check [https://reactjs.org/docs/typechecking-with-proptypes.html] - in Typescript declare proper interfaces
- Mind the rendering [https://reactjs.org/docs/optimizing-performance.html]
- Keep complex animations outside of React's state - DOM it's better in it than Virtual DOM
- Avoid arrow functions and bind in render function ([https://medium.freecodecamp.org/why-arrow-functions-and-bind-in-reacts-render-are-problematic-f1c08b060e36])

Boilerplate

We prepared boilerplates, which will help you with setup new project with React framework. You can find them here:

- TypeScript - https://github.com/SwingDev/react-ts-boilerplate

Linter

Our linter settings for React you can find here:

- TypeScript - https://github.com/SwingDev/react-ts-boilerplate/blob/master/tslint.json

Markup

Background

Even if you work in framework environment, you should keep in mind writing semantic markup. This means that markup should contain semantic elements introduced mostly in [https://developer.mozilla.org/en-US/docs/Web/HTML/Element].

For coding static sites we recommend using any template engine like [http://handlebarsjs.com/] or [https://pugjs.org/api/getting-started.html].

We recommend you to take a look at this [https://github.com/hail2u/html-best-practices] of writing HTML.

Rules

- Use semantic elements whenever you can
- When coding static sites, prefer templates engines over bare HTML
- Declare attributes starting from *class*, *type* and end with *style*
  - always follow linter settings - especially in React or Vue.js environment
- Use double quotes for declaring value of the attribute ([https://github.com/hail2u/html-best-practices#dont-mix-quotation-marks])

- Be consistent with [https://developers.google.com/style/html-formatting]

JavaScript (ES6+ standard)

Background

ECMAScript 6, known as JavaScript ES6 or ES6, is a modern standard of one of the most popular programming languages. It introduces several new APIs, like spread operator, template strings, and others. Initially introduced by Babel transpiler, now it's an industry standard, thanks to good [https://caniuse.com/#search=es6].

Along with Typescript, ES6 is our primary language of choice. It's supported by React and Vue.js frameworks, which we use in our daily basis work.

If you need more informations about ES6 standard, here are couple of materials, that you may like:

- http://exploringjs.com/es6/index.html
- https://www.tutorialspoint.com/es6/index.htm
- https://zellwk.com/blog/es6/

Rules

- Use const over let and var
- Use [https://zellwk.com/blog/es6/#the-rest-parameter-and-spread-operator] for objects and arrays
- Use [https://zellwk.com/blog/es6/#destructuring] to get properties from object / array

- Use arrow functions - especially, when you want to maintain context
- Use [https://zellwk.com/blog/es6/#template-literals] for writing HTML templates in JavaScripts
- Use ES6 classes for defining new objects

TypeScript

Background

> TypeScript starts from the same syntax and semantics that millions of JavaScript developers know today. Use existing JavaScript code, incorporate popular JavaScript libraries, and call TypeScript code from JavaScript. Types enable JavaScript developers to use highly-productive development tools and practices like static checking and code refactoring when developing JavaScript applications.

TypeScript is mainly superset of JavaScript language. This means that you get all this that is offered by JavaScript, plus new features (compatible with ES6 [page 47]) and type checking.

Thanks to TypeScript you can write maintainable and readable code. TypeScript brings types from statically typed languages to JavaScript. You don't need to worry about type mismatching, because compiler catches it for you.

There is one tradeoff, you lose the dynamic nature of JavaScript. Also, type checking is as much accurate as you are precise in types defining. The learning curve is steep for this language, but still, it's worth to use it in your project.

Checkout some tutorials about TypeScript:

- https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html
- https://toddmotto.com/typescript-introduction

Rules

- Disallow usage of *any* type
- Use [https://www.typescriptlang.org/docs/handbook/enums.html] to define flag variables
- Use interfaces over simple types
- Take advantage of using [https://www.typescriptlang.org/docs/handbook/generics.html]
- Consider using linter or / and [https://basarat.gitbooks.io/typescript/docs/styleguide/styleguide.html]

Linter

To use full potential of the TypeScript, we use strict linting rules. This allows us to levarage all the hard work of types checking to the language and helps with development if done properly. The main rule you should obey: The Stricter, The Better.

Our linter settings for TypeScript you can find here - https://github.com/SwingDev/react-ts-boilerplate/blob/master/tslint.json

Styles (SCSS)

Background

> **Cascading Style Sheets** (**CSS**) is a [https://developer.mozilla.org/en-US/docs/DOM/stylesheet] language used to describe the presentation of a document written in [https://developer.mozilla.org/en-US/docs/HTML] or [https://developer.mozilla.org/en-US/docs/XML] (including XML dialects such as [https://developer.mozilla.org/en-US/docs/SVG] or [https://developer.mozilla.org/en-US/docs/XHTML]). CSS describes how elements should be rendered on screen, on paper, in speech, or on other media.

For writing CSS styles we prefer SCSS (Sassy CSS), which is Sass with CSS syntax. Due to the cascading nature of CSS you should use any class naming convention. We prefer BEM, which quick introduction you can read here.

When we make web application using one of our preferred frameworks, we use CSS modules for React applications and Vue.js (or at least scoped CSS classes in this case).

Rules

- Always [https://github.com/necolas/normalize.css/] your styles
- Define [https://seesparkbox.com/foundry/naming_css_stuff_is_really_hard]

- Never write selectors using only tags, attributes and ids
- Use any naming convention (like BEM or CSS modules) to avoid global scope issues
- For styling elements use mostly pseudo-elements instead of *div*
- Mind writing styles for different states of elements, like “hover”, “active” and “focus”
- Define basic styles for typography - titles, default text, links, etc.
- Keep your code flat, don’t nest (visually) selectors no more than [https://medium.com/@mciastek/s-css-best-practices-that-you-have-not-yet-known-ba2f6329b5dd]
- Define variables for common properties like [https://davidwalsh.name/sass-color-variables-dont-suck] or sizes

Linter

Our linter settings for CSS styles you can find here - https://github.com/SwingDev/react-ts-boilerplate/blob/master/.stylelintrc.json

Real-life connections

Background

Majority of people use internet on their mobiles (2018). This behavioral change impacts everything we're doing on the frontend side.

Facts:

- LTE is a horrible connection type
- not all requests go through
- speed is not consistent
- connection might be offline one second and online the next second
- not all responses come through, even though the request actually worked
- backends might be busy one second, and absolutely fine the next one
- LTE works 3x worse outside of Poland, we're lucky :-)

Our stance:

- We accept these facts as facts of life
- We craft our applications in a way that users are not impacted by these facts - we are, they are not
- Users should be able to use our applications in the subway, while driving, because that's what they do

- Users should always know what's going on
- Users don't care about technology - they want to get something done
- Users must not be guinea pigs - they must not be the first to test e.g. how the app behaves

Rules

UI / UX

- Loading state is communicated at all times.
    - all async. actions progress is communicated
        - https://semantic-ui.com/elements/button.html#loading
    - we minimize layout changes between empty state, loading state, error state and 'fully loaded content' state
        - https://semantic-ui.com/elements/segment.html#segment
- App is usable on a bad connection
    - we automatically retry idempotent requests that failed
        - with a timeout
        - with no response
        - with non-critical error (like 5xx status code class)

        - example implementation:
            - https://github.com/SwingDev/blz-web-app/blob/master/client/utils/axiosAutoRetry.js https://github.com/SwingDev/blz-web-app/blob/master/client/services/categories.js
    - we communicate critical errors in a meaningful way and give users the ability to do something about it - like retry the same action
- We interpret errors properly, examples:
    - 404 on a DELETE request is not an error, expected state does match the facts > object does not exist
    - 410 Conflict on a PUT is not an error if the response matches the request
    - think about different scenarios :-)

Development

- We develop with link throttling enabled in the browser - as next to no users have a fiber optics internet connection:
    - https://developers.google.com/web/tools/chrome-devtools/network-performance/network-conditions
- Use throttling to simulate average devices performance
    - https://umaar.com/dev-tips/88-cpu-throttling/
- We develop using a variety of user accounts:

    - with content
        - some
        - a lot
    - without content - try being a new user once in a while, they make the application grow, we care about them :-)
- We test error handling at all times:
    - try accessing a resource that was deleted in a different tab
    - try killing your wifi and going through the application > that’s the best simulation of a user using public transport :-)
- Leverage caching in your application
    - Use application cache
    - Use browser cache [page 60]

Accessibility

Background

Keep in mind that audience of your website can be also people with disabilities. When writing your code remember to use accessibility rules defined by [https://www.w3.org/TR/WCAG20/].

Here are some handful articles about accessiblity in web:

- https://medium.com/alistapart/writing-html-with-accessibility-in-mind-a62026493412
- [https://axesslab.com/alt-texts]

Rules

- Use [https://a11yproject.com/posts/getting-started-aria/] for defining content of the element
- Mind document structure - especially proper heading [http://html5doctor.com/outlines/]
- Use dev tools to check for accessibility issues - [https://developers.google.com/web/updates/2017/05/devtools-release-notes#lighthouse]
- Use [https://axesslab.com/alt-texts] to provide text back-up for figures

Battery performance matters

Background

As we already concluded - majority of our users browse the web on their phones. How about the others? Well - they use laptops. See the common thing here?

Pretty much ALL of your users run on battery power!

Now - battery capacity is very limited, and we did not find any meaningful way of increasing the density of battery cells in the last 10 years.

So how does a MacBook last 12h on battery power? Easy - by switching off hardware that's not used and underclocking the hardware that's barely used.

Our job? Use the least possible resources to get the job done and idle on idle.

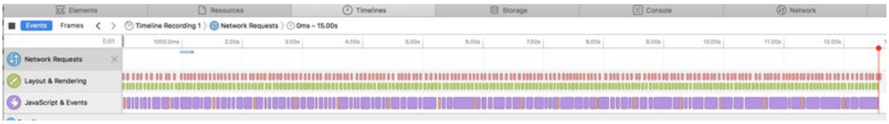
I'm sure you've seen how all the websites whose income is based on ads affect your browser. Don't be that website.

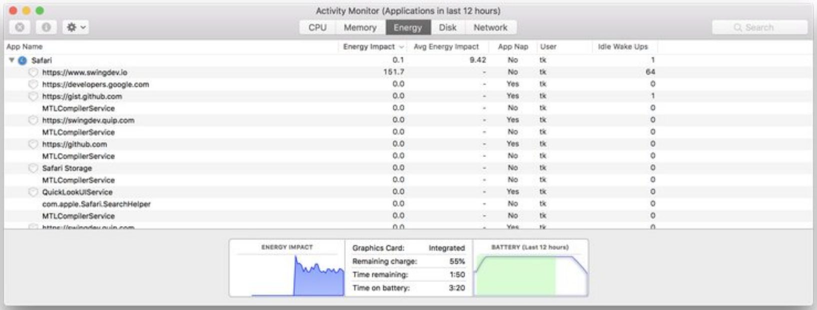
Rules

cpu usage

- Do not use CPU when the website is idle, or use it the least possible way.

- Example of an idle website redrawing all the time (in this case - still redrawing canvas after the animation is finished):




Activity Monitor (Applications in last 12 hours)

App Name	Energy Impact	Avg Energy Impact	App Nap	User	Idle Wake Ups
Safari	0.1	9.42	No	tk	1
https://www.swingdev.io	151.7	-	No	tk	64
https://developers.google.com	0.0	-	Yes	tk	0
https://gist.github.com	0.0	-	Yes	tk	1
MTLCompilerService	0.0	-	No	tk	0
https://swingdev.quip.com	0.0	-	Yes	tk	0
MTLCompilerService	0.0	-	No	tk	0
https://github.com	0.0	-	Yes	tk	0
MTLCompilerService	0.0	-	No	tk	0
Safari Storage	0.0	-	No	tk	0
MTLCompilerService	0.0	-	No	tk	0
QuickLookUIService	0.0	-	Yes	tk	0
com.apple.Safari.SearchHelper	0.0	-	No	tk	0
MTLCompilerService	0.0	-	No	tk	0
https://swingdev.quip.com	0.0	-	Yes	tk	0

ENERGY IMPACT

Graphics Card:	Integrated
Remaining charge:	55%
Time remaining:	1:50
Time on battery:	3:20

BATTERY (Last 12 hours)

Loading performance

Background

Fast application is not only working smoothly during user interaction, but also loads quickly. This means that you should care about how fast your application can be used by user at first time, he enters your website.

There are couple of key metric that you should be aware of:

- First Paint - user sees first element on your website
- First Contentful Paint - user sees first content on your website (article, photo, etc.)
- First Meaningful Paint - user sees [https://developers.google.com/web/fundamentals/architecture/app-shell] (whole layout of your application)
- First Interactive / Time to Interactive - user can interact with your application

More about user-centric performance metrics you can read [https://developers.google.com/web/fundamentals/performance/user-centric-performance-metrics#tracking_fpfcp].

Rules

- Keep your assets as small as possible
    - Minify scripts, styles, images, etc.
    - Load images prepared for certain devices

- Optimize assets loading time
    - Use Cache-Control header on server side - [https://developers.google.com/web/fundamentals/performance/get-started/httpcaching-6]
    - Use Service Workers or App Cache on front-end side
- Prefer lazy loading over eager loading (loading everything at the application boot) - [https://developers.google.com/web/fundamentals/performance/lazy-loading-guidance/images-and-video/]
- Remove all initial render blockers
    - Mostly external scripts / stylesheets placed in <head />
- Inline [https://www.smashingmagazine.com/2015/08/understanding-critical-css/] styles
- Inline critical SVG graphics

How to measure performance of your app?

We recommend using Lighthouse, which makes audit of your website using different tools. It checks performance, accessibility and compatibility with PWA pattern. More about it you can read [https://developers.google.com/web/tools/lighthouse/].

Beside Lighthouse, you should definitely use your favourite browser's dev tools. In Chrome you have access to different tools like:

- [https://developers.google.com/web/tools/chrome-devtools/evaluate-performance/]

- [https://developers.google.com/web/tools/chrome-devtools/network-performance/]
- [https://developers.google.com/web/tools/chrome-devtools/memory-problems/heap-snapshots]

There are other great tools for measuring performance of your application, which you can find in [https://developers.google.com/web/fundamentals/performance/speed-tools/].

What you can do to improve loading time?

Minification

When you see that your application is loading slowly, first thing you need to do it to check if assets are correctly optimised. There are couple ways how you can do it:

- Minify your [https://webpack.js.org/guides/production/#minification] and [https://github.com/webpack-contrib/css-loader#minimize] (using Webpack)
- Optimise you images [page 65]
    - you can use [https://github.com/Klathmon/imagemin-webpack-plugin] for Webpack

Caching

You have your assets optimised, but still you website can load slowly, especially during next reloads. There is one thing that you need to remember - never load the same resource twice. By “the same” we mean, resource that is not changed over time. That’s why you need to leverage browser caching.

Here are some tips how you can do it:

- Use Cache-Control header on server side - see [https://jakearchibald.com/2016/caching-best-practices/]
- Cache files on client with Service Worker - see [https://developers.google.com/web/ilt/pwa/caching-files-with-service-worker]

More about caching you can find [https://developers.google.com/web/fundamentals/instant-and-offline/offline-cookbook/].

Lazy loading

Lastly, it’s good to load only required assets. Imagine that you build huge application. You definitely don’t want to load all assets at the first load. It’s better to lazy load them.

You can lazy load images, styles, scripts and any other files. Here are tools which you can use:

- scripts - [https://webpack.js.org/guides/code-splitting/]
- styles - [https://github.com/addyosmani/critical]
- images - [https://github.com/verlok/lazyload]

PRPL pattern

[https://developers.google.com/web/fundamentals/performance/prpl-pattern/] is a pattern for structuring and serving Progressive Web Apps (PWAs), with an emphasis on the performance of app delivery and launch. It stands for:

- **Push** critical resources for the initial URL route.
- **Render** initial route.
- **Pre-cache** remaining routes.
- **Lazy-load** and create remaining routes on demand.

Image / Video optimization

Background

Images and videos are often the first place to optimise in order to improve your application loading time. You need to remember to not serving source files on production. The same goes for images and videos.

Usually you receive hi-res assets from designer, which can be heavy. As a developer you need to keep the whole site smaller than a few megabytes tops and we have to keep the images crisp on high density displays (like iPhone's Retina display). It's important to find a golden mean between images / videos size and their quality.

More information about image optimisation you can find [https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/image-optimization].

Rules

- Lazy load the right size of the image / video for the user's screen.
- Use [https://developer.mozilla.org/en-US/docs/Learn/HTML/Multimedia_and_embedding/Responsive_images] whenever you can.
- Optimise images and videos in production build

How can one optimize images and videos

Images

- Don't just drop the images you receive right into the folder with application's assets.
  - Have images sized in multiple variants - all sized just right for how big the image will be **in your layout.**
  - Always prefer JPG over PNG - for non-transparent graphics.
  - Use ImageOptim on every image you commit.
  - Recompress JPGs to quality that's good - use imagemagick's 'convert' utility, e.g. like this:

```
convert intro-1280x1280.png -resize 480x -quality 80 tile@1x.
    jpg
convert intro-1280x1280.png -resize 960x -quality 75 tile@2x.
    jpg
convert intro-1280x1280.png -resize 1280x -quality 65 tile@3x
    .jpg
```

Prefer manual approach to an automated solution, as the solution usually won't know what size the image will be displayed at.

Videos

- Videos are huge!
- Embed YouTube / Vimeo if possible.
- Use ffmpeg toolset to recompress the videos, e.g.:

```
ffmpeg -i input.mp4 -y -movflags +faststart -strict
    experimental -c:v libx264 -crf 26 -vf scale=1280:720 -s
    1280x720
```

’`crf`’ flag determines the final quality.

Animations performance

Background

Beautiful design and UX doesn't feel right if it's not accompanied by 60fps buttery smooth animations and scrolling.

Going below 60fps also show inherent, sometimes hidden, issues with the JS main thread and the way we handle the DOM. Striving for smoothness helps us prevent many issues - battery performance, inability to cope with large datasets - before they impact real users.

Rules

- Develop a keen eye for 'jerkiness', non-60fps animations, etc. - and do treat this as an issue, not something to be accepted.
    - https://developer.chrome.com/devtools/docs/rendering-settings
- Jerkiness / pauses are commonly caused by 3 things:
    - bad JS performance, blocking the main thread
        - https://developers.google.com/web/tools/chrome-devtools/rendering-tools/js-execution
    - unnecessary / huge repaints
        - https://developers.google.com/web/tools/chrome-devtools/rendering-tools/

        - https://developer.chrome.com/devtools/docs/demos/too-much-layout
    - reflow
        - https://gist.github.com/paulirish/5d52fb081b3570c81e3a
- Try to measure performance using RAIL model
    - https://developers.google.com/web/fundamentals/performance/rail

How to create smooth animations?

In order to make smooth animations you need to remember one important thing - no matter how complex is your animation, you need to do it under ~16ms. This is not a long time to do all your stuff, so keep your animation optimised as much as possible.

Understand the Event Loop

JavaScript is very different from other language you might know. This affects animations greatly, and the main reason for this is - the event loop. Ever wondered why Promises and Callback Hell is something that only happens in JS? That's the reason.

These are enough to get you started:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop

https://medium.com/front-end-hacking/javascript-event-loop-explained-4cd26af121d4

Animation types

Whenever you start coding animation, first thing you need to do is to define a right tool for the job. The most general types of animation are complex and simple animation.

Simple animation - basic motion, often triggered by pointer event like *hover*

Complex animation - combination of several tweens, usually joined as one scenario

The rule of thumb is to use CSS transitions / animations for **simple animations** and use JavaScript (mostly along with animation library like [https://greensock.com/gsap] or [http://animejs.com/]) for **complex animations**.

Avoid repaints and reflows

Mostly it doesn't matter if you animation is a **simple** or **complex** one. What matters it that if it triggers repaints or reflows.

Reflow (layout)

> Layout (or reflow in Firefox) is the process by which the browser calculates the positions and sizes of all the elements on a page.

Repaint

> Paint is the process of filling in pixels. It is often the most costly part of the rendering process. If you've noticed that

your page is janky in any way, it's likely that you have paint problems.

If you have any problems with animation performance, it's highly possible that your animations cause repaints or reflows. To check what properties can trigger these phases, we suggest you to take a look a super-handy [https://csstriggers.com/] site.

Using FLIP technique

There is one helpful trick that you can use to improve your animations' performance. It's called FLIP, which stands for First, Last, Invert, Play. This technique simply puts the computation phase at the animation's beginning, so all the jerkiness-possible stuff is done at the bootstrap.

First

the initial state of the element(s) involved in the transition.

Last

the final state of the element(s).

Invert

here's the fun bit. You figure out from the first and last how the element has changed, so – say – its width, height, opacity. Next you apply `transforms` and `opacity` changes to reverse, or invert, them. If the element has moved 90px down between First and Last, you would apply a transform of -90px in Y. This makes the elements appear as though they're still in the First position but, crucially, they're not.

Play

switch on transitions for any of the properties you changed, and then remove the inversion changes. Because the element or elements are in their final position removing the transforms and opacities will ease them from their faux First position, out to the Last position.

More about FLIP you can find [https://aerotwist.com/blog/flip-your-animations/].

Fresh data should propagate across the whole UI

Background

One of the tell-tale signs of well-designed data handling layer: fresh data propagates across the whole UI without having to maintain all the links manually.

From the user's perspective - if I save my To Do item I expect this change to propagate automatically to all other pieces of the UI.

From the developer's perspective - I expect it to just work if I add a new UI component or if I add a new place that allows me to edit this To Do item.

If I don't have to know the implementation details behind the propagation - I won't forget to add something I didn't know I had to add in the first place :)

Rules

- All UI components must respond to changes in the underlying data, or force a refetch any time data it's showing is stale.
- Persistence layer should be propagating the committed changes automatically to where it matters.
    - This means that there must also be a way to 'stage' changes before they are saved. I don't want my draft state to prop-

agate everywhere - 'cancel' functionality would be a hell to implement

- Persistence layer could also - instead of propagating the changes - mark the objects as 'stale' - so that if there is a component that needs this data - it'll force a refetch.

How to test this behavior?

Always check your UI components for their behavior when underlying data is changed by another component.

Make sure this logic is not mixed up with the UI layer - it should be reasonable transparent, so that a new person can add a new UI component easily, without having to know anything.

Check the behavior when the changes are cancelled (as in hit 'cancel' in the object edit component). These should not be propagated.

Check the behaviors of lists when an object is added / removed. They should also update automatically!

Check the behavior of components that show the object whenever this object is removed while the component is shown.

Check if your 'refetch-if-stale' implementation doesn't cause too many refetches for the same object. Always batch them up.

Implementation hints

global cache

- Implement a global state using per-item caches, populated by events sent from the application's persistence layer.
- Marking data as to-be-refetched when actions could have changed it (e.g. parent entity aggregations when changing child entity).
    - Persistence layer to refetch the object instead of giving the cached one when requested.

Handle conflicts consciously, not accidentally

Background

So what's a conflict? - in our case conflict is said to happen when our application tries to update an object which state changed in between the last fetch and now. It might have been edited, it might have been removed.

You might think that this is an edge case - **it's not**. Your application will be used by multiple people on the same account, by the same user in multiple tabs, by the same user on different devices at the same time. You might want to add live websocket connection to propagate changes as they happen.

All of these cause conflicts.

Rules

- Always make a decision how this particular entity / action should handle conflicts. Have this, and the reason why, ready for your PR review.
- Always make a decision whether this particular entity in this scenario should PUT or PATCH the changes.
- Solve conflicts on the frontend automatically whenever possible.
    - Some UI actions should automatically overwrite all the

fields **that were explicitly changed by the user.** Example - JIRA task title. I don't care that someone else changed the title or description in the meantime - if I wanted to change the title, I still want it to happen without any additional alerts. Now - for description it would be the opposite - as it might have taken a long time for someone else to put their changes in.

- Some UI actions should automatically overwrite the entire entity. Example - background worker task which job is to update the object from a 'ground truth' - e.g. external API.
- Some UI actions should just fail whenever the original object changed. Example - Plane seat booking - if it's been snatched by someone else there is nothing we can do.
- Some failures are not failures, and should not be treated as such:
  - 404 on a DELETE action - state is exactly as expected, action was a success
- Some things must be left for a user to fix - make sure it's a decision though, not an accident.
- Balance occurrence rate + frustration vs time to implement on edge cases:
  - Example 1 - while PATCHING the object I receive a 404
    - Option 1 - alert the user and transfer his data to 'add object' UI dialog.
    - Option 2 - automatically create a new object

        - Option 3 - just alert the user and have him retype everything somewhere else - **don’t be this guy! - unless you have a valid reason.**

- Always handle this automation after a failure returned by the back-end - usually it will be 409. Don’t leave 409 unhandled.

- Handle changes being made in the background to the entity the user is editing on your ’edit’ component.

    - Example - I open a JIRA task panel, start editing. In the meantime another polling component fetches the new version of this object. I want this to be handled.

How to test this behavior?

Explicitly check this behavior e.g. by using your component in to tabs or on two devices at the same time.

Implementation hints

Conflict detection

If there is an autoincrementing ’version_id’ field - remember to send it to the backend alongside the changes.

How to be able to make this kind of decisions in your error handling logic

Your ’draft’ objects implementation should have the ability to check which fields were changed by the user, and which were not. This gives you a lot of power.

Prefer PATCHing diff over PUTting the whole entity.

Care about state not single actions

- “I want this object to be gone” not “I want deletion to succeed”
- “I want to change this field to X” not “I want the PATCH operation to succeed”
- Add logic to handle success / failure of your methods independent of actual underlying action success - e.g. 404 on DELETE is a success - object is gone.

Usual conflict handling scenario

- Keep your 409 handler separate.
    - sometimes: entity-wide
    - sometimes: action-wide
- In your 409 error handler:
    - Download new state, keep previous diff
    - Try to resolve conflict automatically:

    - Option 1) diff user made can be patched onto new state without conflict
    - Option 2) we know by design this action should overwrite the state
- Alert user if not the case, and let him decide

Lazy-load in the UI

Background

To fulfill our requirements for data handling all UI components need to be ready to lazy-load content, even if application doesn't call for it (e.g. because the data is always there - will it be, always? nope).

It's for a simple reason - whether the data should be fetched, refetched or served from cache is not a decision that should be made by the UI component. It belongs to the persistence layer.

Now - if your component is ready to lazy load content - it'll be robust enough to handle all the other async scenarios too, for free - e.g. background changes to the object, action failure scenarios - the lot.

Rules

- Component is lazy-loading ready, when:
    - it calls for the persistence layer to fetch the object when it's shown
    - it handles all entity loading / error states
    - it shows updated content when the entity updated in the background - not just this once, but always
        - exception - 'edit' components - they should resolve conflicts as well as update the changes

- Lazy-load and create remaining routes on demand (*L* from [https://developers.google.com/web/fundamentals/performance/prpl-pattern/])

How to test this behavior?

Check your component's behavior when the entity is changed by another component on the screen. If an entity is marked as stale by another component - will it show new content? Test the loading state and the error handling.

How to preload routes on demand?

The first load of your application is significant. Ideally, during this 2-3 seconds, you need to keep the user on your site, so it's better to load the content fast. In case of small applications, this seems to be an easy task, but what happens, when your application's size increases? It's good to keep things small. Remember, 100kb of JavaScript isn't the same as 100kb of JPEG image.

To reduce the [https://medium.com/dev-channel/the-cost-of-javascript-84009f51e99e], split your code into several chunks and load them, when necessary. You have plenty of tools, which can help you with it. With Webpack, you can use [https://webpack.js.org/guides/code-splitting/#dynamic-imports]. In React with React Router, you can [https://reactjs.org/docs/code-splitting.html].

Batch loading

Background

Many components on the page might require access to the same entity. This cannot cause multiple fetches, as it’s a waste of bandwidth and time.

Same goes for a lot of individual objects - if we’re showing 50 entities in a grid - they must be fetched in bulk, not doing 50 separate requests.

Rules

- Batch loading of the same entity
- Batch loading of multiple entities at the same time whenever possible.
    - Always question loading objects one by one if showing more than one at a time.

How to test this behavior

Always test your component and application behavior by checking out ’networking’ tab of your Dev Tools.

Be able to explain each and every request made by your application - and make them a decision, not an accident.

Make it a job of your persistence layer to coordinate loading.

Implementation hints

Batch requests for the same entity

Cache promises in the persistence layer. Return the same promise if available. Decide if refetch necessary there.

Batch requests for multiple entities

Always check if the API allows for batching GETs - it usually does. Demand it from your Backend team if needed.

In the persistence layer - gather IDs to be fetched for a single 'tick'. Then gather them up into a single request, and resolve all of the promises after fetching.