Architecture Reconstruction: Zeegu-React

Software Architecture, MSc (Spring 2024)
KSSOARC2KU

Bjarke Brodin Larsen – bjal@itu.dk

May 29, 2024

Contents

1 Introduction

Zeegu is a language learning assistance tool that helps users learn by reading and revising previously read language on a word-by-word basis. It contains serveral features such as tracking progress and providing material for language learning with on demand translations. It is also meant to offer some functionality to teachers interacting with the material provided, but I have not probed this part of the app much and leave this mostly out of scope.

The front-end of Zeegu is the subject matter of this paper. More specifically, I attempt to recover information about its architecture, and subsequently reflect on the results of said recovery.

2 Methodology: tooling and process step-by-step

In the following section I will describe which tools I deployed in what order and why I chose to do so.

2.1 Getting acquainted with the system – ChatGPT

Because getting hints and ideas about the elephant we are attempting to map out is not dependent on precision and in the beginning is mostly a text-parsing task, I utilized ChatGPT[1] to summarize the contents of the repository and get an introductory idea (see Appendix A). Because ChatGPT can search the web it surprised me by being adept at suggesting tools for further inspection which might have come in handy!

2.2 Getting a more exact feel of function – running the system

After quickly getting some hints and a notion about what system I am dealing with, I proceed to run the system using the online beta deployment hosted at https://zeeguu.com. This is mostly to probe the system functionality and improve my mental map of how the codebase manifests to the user.

2.3 Researching and briefly parsing the code manually

By reviewing and researching the React framework on which the subject codebase is built, and subsequently simply looking at the code. I attempt to acquire a feel for how the code fits together, how are string translations handled, how are components themed and global variables managed, how is routing done.

2.4 Mining data and producing a module view – python3, networkx, pyvis

With a pretty good idea about the overall system structure, I begin semi-automatically attempting to map the codebase using python. Code can be found at https://github.com/bjarkebrodin/recovery. I do this by first extracting the imports between files (see Figure 2) then subsequently I abstract to top-level modules and consider only imports in between such modules (see Figure 3 and Figure 4). This approach is very similar to the one demonstrated by Mircea Lungo in the reconstruction lectures[2]. These networks are visualized dynamically as html files and the reader can inspect them at will from the repository, to make the data inspectable they are programmed with tooltips that show detailed information about nodes (see Figure 6), and edges (see Figure 5). I visualize the gathered data and attempt to infer interesting facts about the architecture. The outcome of this step is described in more detail throughout the remainder of this paper. I use the tools `pyvis`[3] and `networkx`[4] to help me do this without implementing graphs and visualisation myself.

2.5 Superficially assesing external dependency health – npm audit

As a finishing step, only because it is so easily available in npm projects: to quickly ascertain the health of external dependencies used I run `npm audit` to get an idea about how much (potential) debt there might be in the way external dependencies are used.

3 Results

Through semi-automatically eliciting an approximate dependency (JavaScript `import`) structure of the code-base (as described in Section 2.4), I produce a module view of the Zeegu front-end (see Figure 1).

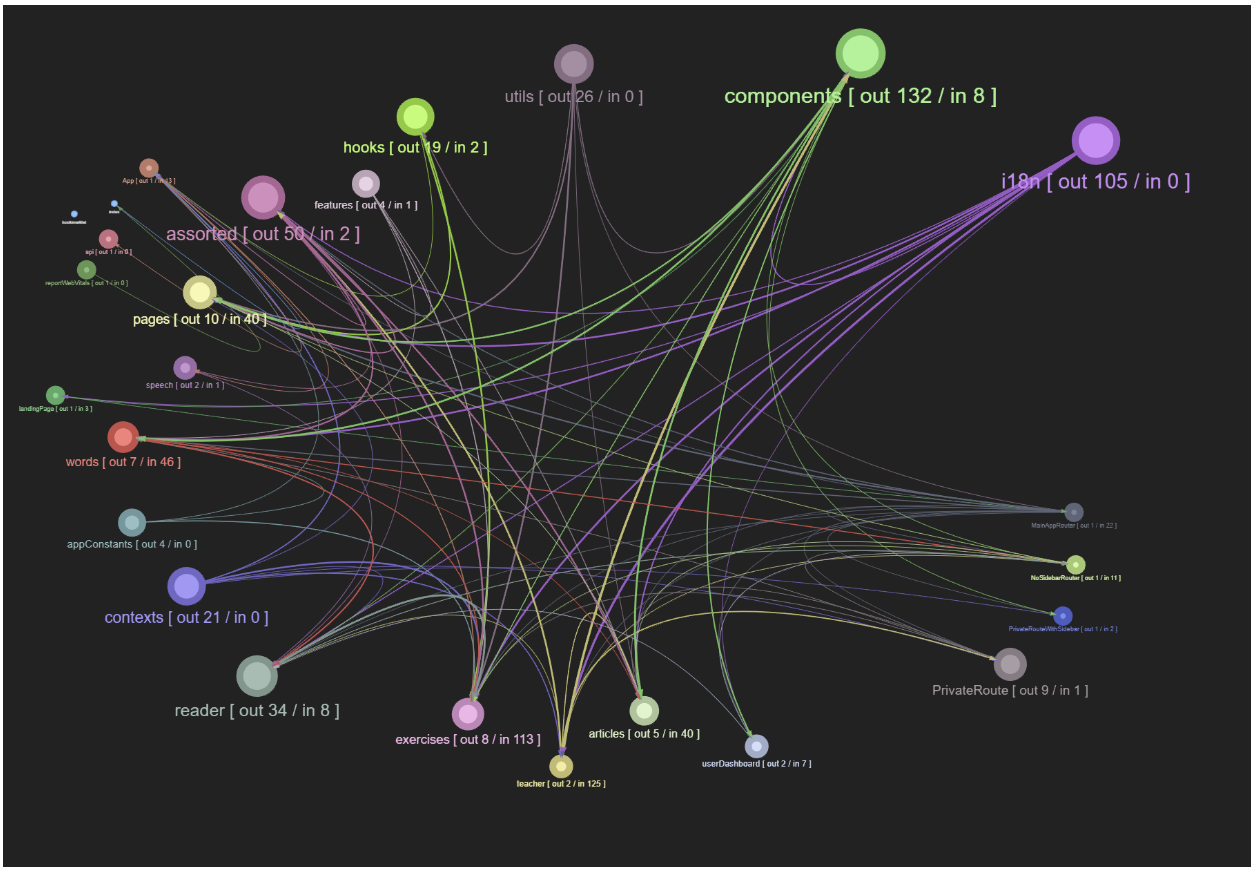

Figure 1: Cleaned and colored visualisation of all import statements abstracted to top-level modules. Thicker edges represent more imports (logarithmically scaled) and edge direction represents which what is being imported to where. Colors do not have meaning but are meant to help better decode the network by eye. Modules are appended with information about how many times they are imported and how many times they import another module. A bigger circle means a module is imported more times by other modules (*out* value), i.e. might be important or central to the architecture.

> *out* – times imported in a different top-level module
> *in* – times importing a from different top-level module

To frame the information presented in Figure 1 and prime the reader for a brief discussion of this data (Section 4), I begin by briefly outlining and summarizing the degree of interest of some of the most referenced/central modules, in Section 4 I go on consider some of the most interesting areas of this list.

- `utils` – A collection of project local utility code. It has no dependency on other internal modules, however, there may be something to be said about its submodules `utils.routing` and `utils.cookies`.
- `hooks` – A module of various react hooks supporting ad-hoc stateful component enhancement. This is slightly interesting because somewhat random functionality seems grouped together, if nothing else only based on a react convention to have a module to dump all the hoooks in.
- `contexts` – Much like the `hooks` module, this module contains injectable "contexts", a react specific way to do inversion of dependency with a provider/consumer paradigm.
- `components` – (Supposedly) A module of the general UI components used in the frontend, at first glance it is very interesting that components seems to import from both `reader`, `i18n` and `exercises` – there are also 2 imports from `utils` but this is just common code so not of concern.
- `PrivateRoute`, `MainAppRouter`, `PrivateRouteWithSidebar` – Modules for navigating the different parts of the site, these should arguably be encapsulated in a `routing` top-level module and could then be decoupled by fetching routes by lookup instead of reference. It is however not that bad to couple routing tightly as it is mostly expected in a react app of this size and state in development.
- `i18n` – This module contains the configuration of the **`i18n`** module for string internationalization, meaning mostly a collection of strings in different languages to be used elsewhere. Since it only exports and doesn't import from any other local module this is mostly uninteresting, however, there may have been a benefit to adding a layer of indirection between how translations are implemented and the abstraction of fetching strings by lookup. There is also a bit of funny business regarding how non-global language is handled, it seems to be somewhat scattered around the codebase.
- `articles` – This module contains code related to showing, finding and working with articles on the Zeegu platform. It exports to only `teacher` and routers making it seem relatively well-encapsulated.
- `words` – Contains code for the words page and functionality for working with words. The module is relatively uninteresting except for the fact that it has cross-dependencies with `exercises`.
- `teacher` – This module contains functionality for teachers, it almost exclusively imports from `i18n` and `components` and is only imported in routers, making it quite uninteresting.
- `pages` – This module contains the markup and some components used for some of the pages of the app, not all though, some pages are in `landingPage` and again some other pages are elsewhere.
- `reader` – This module implements the article reader and is thus a quite central part of the app, it is however at first glance quite well decoupled and logically encapsulated nicely, meaning initially, not very problematic or interesting structurally.
- `assorted` – This module contains miscellaneous utility code and a single hook, it is opaque to me why this code is not in `utils` and `hooks` instead.

Note that to limit the scope I have somewhat arbitrarily elected a size/interest cutoff of which top-level modules to include in this list, otherwise there would simply be too much to cover. Even discussing all the modules I list above in more depth would be relatively plentiful, thus, I cover only some of the more appearant and superficial concerns in the following section.

4 Discussion

From visualising and inspecting the data more closely (see Figure 1), both by looking at the overview but also by interacting with the generated html to inspect exactly what is imported in which module, I consider a few aspects of the system.

Starting off, there seems to be several top level modules depending on eachother indicating a tighter degree of coupling in between them than we would ideally like:

- `exercises` ↔ `components`
- `exercises` ↔ `words`
- `reader` ↔ `components`
- `routers` ↔ `*` (transitively)

Additionally, it seems less that ideal at first glance that some utility code is in `utils` and some other utility code is in `assorted`, there might be an intention behind this but it would also be explainable by different contributers using different modules for their helper-code (I did not check this in the git files). Either way it seems easy to refactor this to enhance overall structure.

The `hooks` module seems to group code by which category of react abstraction it uses. While this is probably quite conventional, I would argue that there would be a point to having hooks implemented closer to the code that they use. As an example of my point, `hooks.useExtensionCommunication` is essentially just a hook-based interface for the code implemented in `utils.extension.extensionCommunication`, thus I would consider moving the hook to `utils.extension.useExtensionCommunication` (and similarly for other hooks). This would clean the top-level scope and would make sense since when crawling backwards from where the hook is imported would bring you straight to both files, potentially making it less opaque to debug and understand.

Many more things could be said about the system, but for the sake of brevity I shall end my considerations here. Overall, even though a few top-level modules are tightly coupled, the coupling does not seem particularly tight which is positive. Appropriate react abstractions seem used reasonably well (hooks and contexts) and translations are injected from a single point (even if the coupling to the **`i18n`** library is a bit tight). The top-level structure could be more intentional, providing developers with a better idea of how new code is categorized and assistance in forming a mapping between the visual/navigable structure of the app and the structure of the code. By this I mean a clear way to guess exactly where to find the code responsible for rendering what is currently visible.

I would have liked to produce some metrics, especially link depth and cyclomatic complexity (as ChatGPT suggested, see Appendix A p.6) but this did not work at first attempt and I was unable to invest sufficient time in getting working. This would have been an interesting addition to the import structure! I suspect especially cyclomatic complexity would be quite high because of the degree of coupling suggested by the module view.

References

[1] OpenAI, "Chatgpt (20-05 gpt-4o)," https://chat.openai.com, 2024.

[2] M. Lungo, "Lecture notes on architectural reconstruction," April 2024. [Online]. Available: https://github.com/mircealungu/reconstruction

[3] G. Perrone, J. Unpingco, and H. minn Lu, "Network visualizations with pyvis and visjs," 2020.

[4] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using networkx," 2008.

A ChatGPT Transcript

OpenAI

Zeeguu Web Frontend Summary

Here is a link https://github.com/zeeguu/web to a github repository containing the code for a software system, can you summarize the systems functionality, context and architecture for me?

The GitHub repository "zeeguu/web" contains the frontend code for the Zeeguu web application, which focuses on language learning and reading improvement. The system is built using React and follows specific coding conventions such as separating styled components from React components and avoiding hardcoded string constants. The project was bootstrapped with Create React App and expects the `` `/teacher-dashboard` `` to be deployed locally. The architecture emphasizes readability and maintainability with clearly defined coding standards and environment configurations.

For more details, visit the repository on GitHub.

That is excellent - can you expand on the architecture?

Zeeguu Web Architecture

Frontend:

- **React-based**: The application is built using the React library for creating interactive UIs.
- **Create React App**: Initially bootstrapped with Create React App, providing a standardized project structure.
- **Styled Components**: Uses styled-components for applying CSS styles, separating them from the logic in React components.
- **State Management**: Likely utilizes React's context or state management libraries (like Redux) for managing global state, though not explicitly mentioned.

Coding Conventions:

- **Code Readability**: Emphasizes clean code practices, avoiding hardcoded string constants by defining them in separate files.
- **Component Structure**: Components are organized in a way to separate concerns, ensuring that styling, logic, and presentation are distinct.

Environment Configuration:

- **Development**: The repository includes configurations to facilitate local development, including environment variables and scripts for running the application locally.

Deployment:

- **Local Deployment**: The `/teacher-dashboard` endpoint is mentioned, suggesting a component specifically designed for educators, possibly indicating role-based access or specialized dashboards.

Backend Integration:

- **API Consumption**: Although the repository is for the frontend, it likely interacts with a backend API (presumably the main Zeeguu API) to fetch and manipulate data.

For further details, you can explore the code and documentation on the Zeeguu web repository.

Ok that's good, can you identify any design patterns used?

Based on the project structure and code examples from the Zeeguu web repository, several design patterns are evident:

1. **Component-Based Design**:
    - **React Components**: The entire application is built using React components, a core design pattern in React for creating reusable UI elements.
    - **Container-Presentational Pattern**: Likely used to separate components that handle data fetching (containers) from those that handle rendering (presentational).

2. **Styled Components**:
    - **Styled Components**: Utilizes the styled-components library for defining component styles within JavaScript, promoting modularity and scoped styling.

3. **Context API / Redux**:
    - **State Management**: Though not explicitly mentioned, applications of this size typically use the Context API or Redux for managing global state, following the Flux architecture pattern.

4. **Environment Configuration**:
    - **Environment Variables**: Usage of environment variables for configuration management, following the 12-Factor App methodology.

5. **Code Organization**:
    - **Separation of Concerns**: Clear separation between different types of files (e.g., components, styles, utilities) to maintain modularity and readability.

These patterns collectively contribute to a scalable, maintainable, and modular architecture for the Zeeguu web application.

Nice, can you identify any gang of 4 design patterns?

Analyzing the Zeeguu web repository, a few Gang of Four (GoF) design patterns can be identified:

1. **Observer Pattern**:
    - Used in React's state management where components react to state changes, typical in applications with global state management using Context API or Redux.
2. **Composite Pattern**:
    - The component hierarchy in React often follows this pattern, where complex UIs are built from simpler components.
3. **Singleton Pattern**:
    - Likely used for managing a single instance of global state or configuration settings, such as in a Redux store or a configuration file.
4. **Factory Pattern**:
    - Possibly used for creating instances of components or services, abstracting the creation logic.

These patterns help in organizing the code, making it more maintainable and scalable.

How does the application connect to the backend?

The Zeeguu web application connects to the backend via API calls. This is typically done using JavaScript's `fetch` method or libraries like Axios within React components. The API endpoints provide data required for the application, such as user information, language learning resources, and progress tracking. These API calls handle CRUD operations (Create, Read, Update, Delete) to interact with the backend services, ensuring data synchronization between the frontend and backend. Environment variables are often used to manage API base URLs and keys securely.

For more details, visit the Zeeguu web repository.

What are the main and most important components?

The main and most important components of the Zeeguu web application include:

1. **HomePage**: The entry point of the application, providing an overview and navigation to various features.
2. **Dashboard**: Personalized space for users, showing progress, statistics, and personalized recommendations.
3. **Reader Component**: Central to the language learning experience, allowing users to read and interact with content.
4. **Exercise Component**: Offers various language exercises to reinforce learning.
5. **TeacherDashboard**: Specialized component for educators to track and manage their students' progress.

These components interact with the backend to fetch and display relevant data, ensuring a seamless user experience. For more details, visit the Zeeguu web repository.

Can you give me a more detailed description of the functionality of Zeegu?

Zeeguu is a web application designed to enhance language learning and reading skills. Here are the key functionalities:

1. **Personalized Learning**:
    - Tracks individual progress and adapts content based on the user's proficiency and interests.
2. **Interactive Reading**:
    - Allows users to read articles in their target language with integrated tools for translating and saving unknown words for later review.
3. **Exercises**:

- Provides various exercises, such as vocabulary and comprehension tasks, to reinforce learning.

4. **Teacher Dashboard**:

   - Enables educators to monitor and manage their students' progress and activities.

5. **User Management**:

   - Handles user registration, authentication, and profile management.

The frontend, built with React, communicates with the backend via API calls to fetch and update data.

For more detailed information, you can visit the Zeeguu web repository.

Very good, can you analyze the metrics of the repository code? such as link depth and cyclomatic complexity

To analyze the metrics of the Zeeguu web repository code, such as link depth and cyclomatic complexity, follow these steps:

1. **Link Depth**:

   - **Definition**: The number of levels of dependencies in the code. It measures how many layers deep the function calls or module imports go.
   - **Tool**: Use static analysis tools like ESLint, Webpack, or dependency graph tools to visualize the link depth.

2. **Cyclomatic Complexity**:

   - **Definition**: A measure of the number of linearly independent paths through a program's source code. It indicates the complexity of the code.
   - **Tool**: Use tools like ESLint with complexity rules, SonarQube, or JSComplexity to measure cyclomatic complexity.

Steps to Analyze Metrics

1. **Set Up Static Analysis Tools**:

   - **ESLint**: Configure ESLint with rules for complexity and import/export depth.
   - **SonarQube**: Integrate SonarQube for a detailed analysis of code quality, including cyclomatic complexity.

2. **Run Analysis**:

- Execute the static analysis tools on the codebase.
- Review the generated reports for insights on link depth and cyclomatic complexity.

3. **Interpret Results**:

   - **Link Depth**: Identify deeply nested dependencies and refactor to flatten the structure where possible.
   - **Cyclomatic Complexity**: Simplify complex functions by breaking them into smaller, more manageable pieces.

Example Analysis with ESLint and JSComplexity

ESLint Configuration:

```json
{
  "rules": {
    "complexity": ["error", 10],
    "import/no-dependencies": ["error", { "maxDepth": 3 }]
  }
}
```

JSComplexity Setup:

```bash
npx complexity-report -p src/
```

These steps will help you understand the code complexity and structure, allowing you to make informed decisions about refactoring and optimization.

Can you generate a UML class diagram of the codebase?

Generating a UML class diagram for the Zeeguu web codebase involves several steps:

1. **Set Up and Analyze the Codebase**:

   - Clone the repository locally: **`git clone https://github.com/zeeguu/web.git`**
   - Use a tool like Visual Studio Code with extensions such as "TypeScript UML" or "PlantUML" to analyze the project structure.

2. **Install Dependencies**:

- Ensure you have all dependencies installed: `npm install`

3. **Identify Key Components**:

   - Focus on key components and their interactions:
     - `App.js`
     - `HomePage.js`
     - `Dashboard.js`
     - `ReaderComponent.js`
     - `ExerciseComponent.js`
     - `TeacherDashboard.js`

4. **Generate UML Diagram**:

   - Use a UML tool or script to generate the diagram. For example, using PlantUML, you can create `.puml` files and generate diagrams.

Example PlantUML Script

Create a file named `classes.puml`:

```plaintext
@startuml
!define RECTANGLE class
RECTANGLE App {
  + render()
}

RECTANGLE HomePage {
  + render()
}
App --> HomePage

RECTANGLE Dashboard {
  + render()
}
App --> Dashboard

RECTANGLE ReaderComponent {
  + render()
  + translateWord(word: String)
}
Dashboard --> ReaderComponent

RECTANGLE ExerciseComponent {
  + render()
  + checkAnswer()
}
Dashboard --> ExerciseComponent

RECTANGLE TeacherDashboard {
  + render()
  + viewStudentProgress()
}
App --> TeacherDashboard

@enduml
```

Generate Diagram

- Use a PlantUML viewer or command line tool to generate the diagram:

```bash
plantuml classes.puml
```

This will produce a UML class diagram reflecting the primary components and their relationships in the Zeeguu web application. For more detailed and accurate diagrams, you might need to adjust and expand the script based on a deeper analysis of the codebase.

Do you know of any other tools that might help me reconstruct the architecture of Zeegu?

To reconstruct the architecture of Zeeguu, you can use several tools that facilitate understanding and visualizing code structure and dependencies:

1. **Visual Studio Code Extensions**:
    - **JavaScript and TypeScript UML**: Automatically generate UML diagrams from your code.
    - **PlantUML**: Create and preview UML diagrams directly in the editor.
2. **Static Analysis Tools**:
    - **ESLint**: Analyze code for style and potential errors.
    - **SonarQube**: Comprehensive static analysis tool that provides insights into code quality and architecture.
3. **Dependency Visualizers**:
    - **Webpack Bundle Analyzer**: Visualize module dependencies and sizes in a web project.
    - **madge**: Generate visual dependency graphs.
4. **Code Documentation Tools**:
    - **JSDoc**: Generate documentation from comments in your JavaScript code.

Using these tools can help you gain a deeper understanding of the codebase and reconstruct its architecture effectively.

B Approximate Time Allocation

- Development & tool learning: 55%
- Researching react: 5%
- Playing with/wrangling data: 10%
- Reflecting on results: 5%
- Writing: 25%

C Data Visualisation

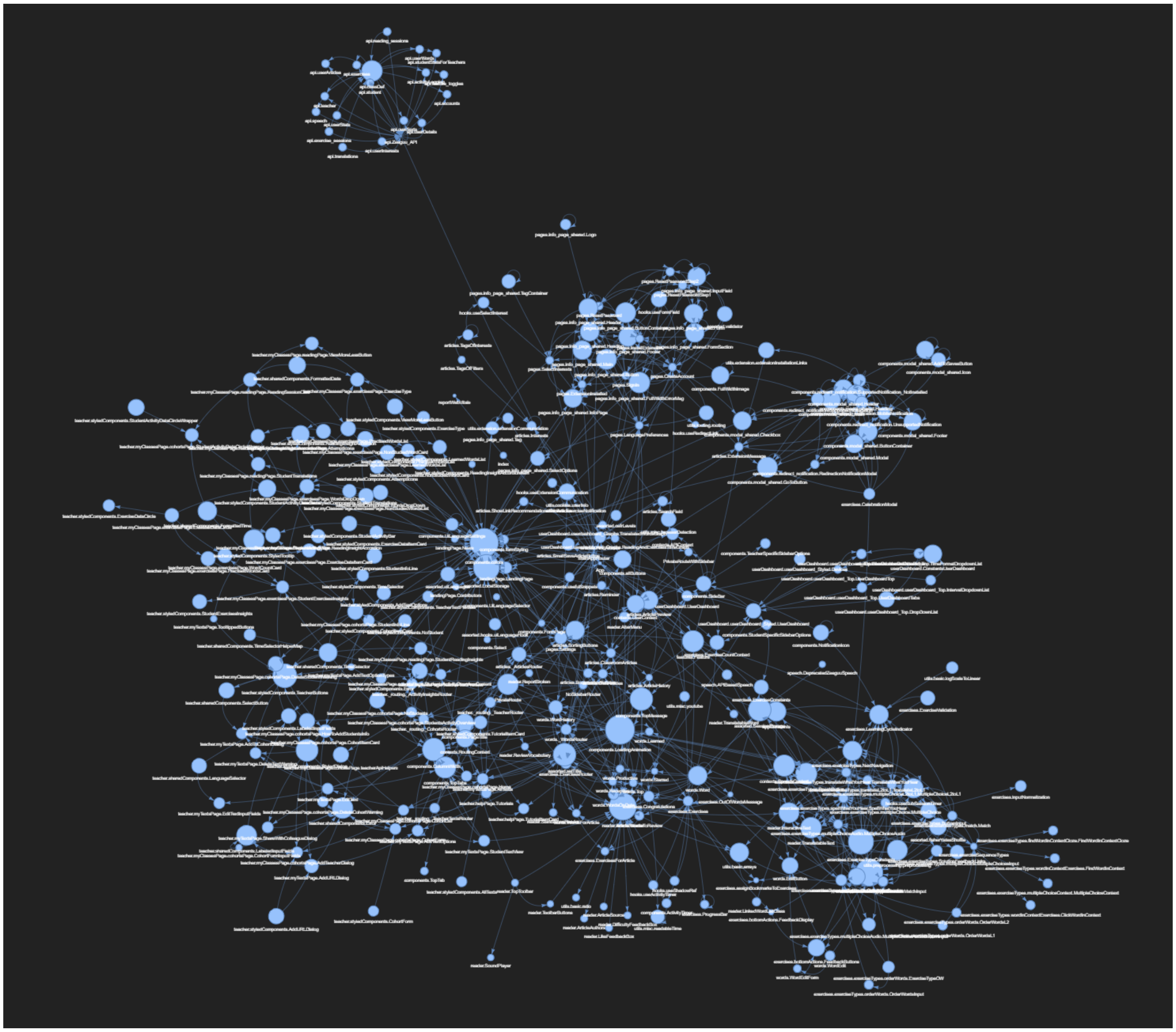

Figure 2: Unfiltered visualisation of all import statements in the sourcecode

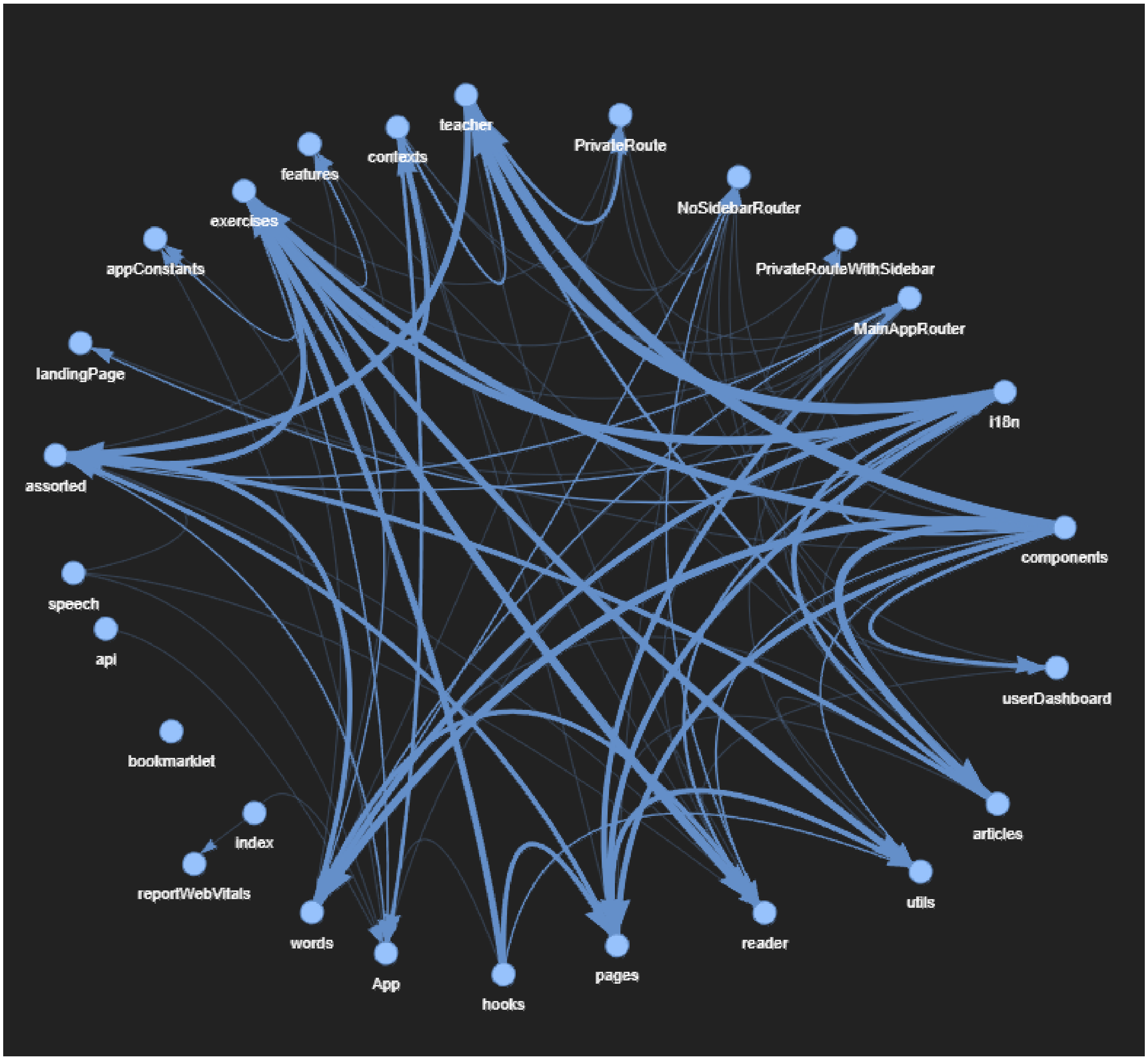

Figure 3: Visualisation of all import statements abstracted to top-level modules. Thicker edges represent more imports (logarithmically scaled) and edge direction represents which what is being imported to where.

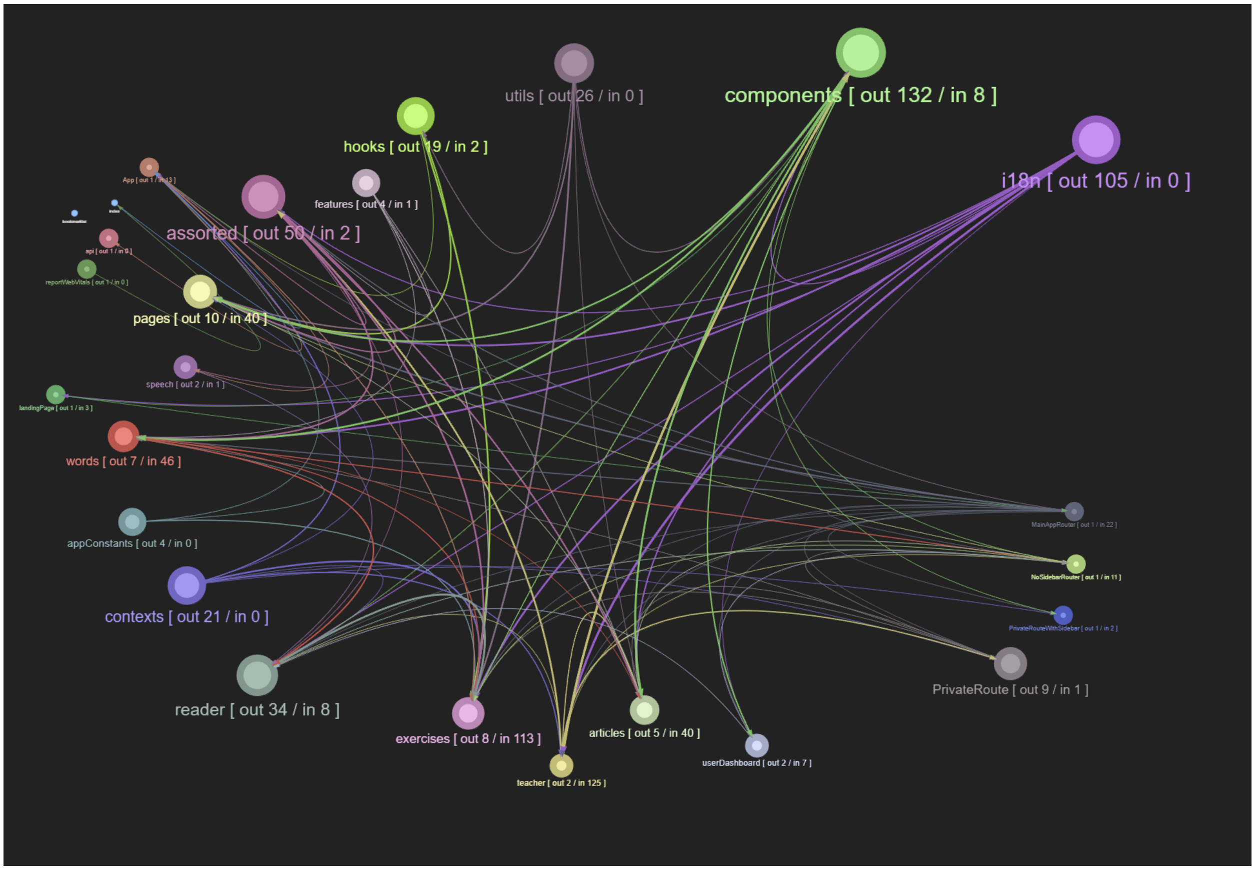

Figure 4: Cleaned and colored visualisation of all import statements abstracted to top-level modules. Thicker edges represent more imports (logarithmically scaled) and edge direction represents which what is being imported to where. modules are appended with information about how many times they are imported and how many times they import another module:

out – times imported in a different top-level module

in – times importing a from different top-level module

Figure 5: Tooltip displayed on edges when visualisation is inspected dynamically as .html

out – times imported in a different top-level module

in – times importing a from different top-level module

module: teacher
TOTAL [out 2 / in 125]
utils [out 0 / in 0]
i18n [out 0 / in 47]
hooks [out 0 / in 0]
landingPage [out 0 / in 0]
appConstants [out 0 / in 0]
reportWebVitals [out 0 / in 0]
contexts [out 0 / in 3]
features [out 0 / in 0]
MainAppRouter [out 1 / in 0]
teacher [out 0 / in 0]
assorted [out 0 / in 13]
PrivateRoute [out 0 / in 4]
bookmarklet [out 0 / in 0]
components [out 0 / in 54]
index [out 0 / in 0]
userDashboard [out 0 / in 0]
App [out 0 / in 0]
words [out 0 / in 0]
articles [out 0 / in 3]
PrivateRouteWithSidebar [out 0 / in 0]
speech [out 0 / in 0]
reader [out 0 / in 1]
api [out 0 / in 0]
NoSidebarRouter [out 1 / in 0]
exercises [out 0 / in 0]
pages [out 0 / in 0]

Figure 6: Tooltip displayed on nodes when visualisation is inspected dynamically as .html
out – times imported in a different top-level module
in – times importing a from different top-level module