React-bratus: Visualising React Component Hierarchies

Stephan Boersma, Mircea Lungu
IT University of Copenhagen
Copenhagen, Denmark
{step,mlun}@itu.dk

***Abstract*—We report on the development of *react-bratus*, a prototype providing developers with an interactive web-based component hierarchy visualization for React-based projects. We continuously evaluate *react-bratus* in multiple iterations with junior developers and find that component hierarchy visualizations are a promising tool for helping novice developers reverse engineer React-based applications.**

***Index Terms*—react.js, component hierarchy, polymetric views**

I. Introduction

React.js[1] is a widely used JavaScript library that enables developers to build user interfaces for the web[2]. React.js is component-based, allowing developers to create components with individual state management and business logic. The focus on componentization of the UI has, as a result, the fact that React.js applications can grow large in size and complexity, easily arriving at hundreds of components.

Applications of this complexity can be challenging for developers to reverse engineer and maintain [1]. To our knowledge, no research has been done until now in supporting developers with gaining an overview of their React.js-based applications.

Given that the component tree is the main axis of decomposition of a React application, in this paper we propose *react-bratus*, a prototype developer tool that provides developers with an interactive web-based component hierarchy visualization. *react-bratus* is being developed using a methodology named design thinking which encourages early and continuous involvement of users, in our case, junior software developers aiming to understand a new React codebase.

The contributions of this paper are: (1) a study of developer needs for reverse engineering React applications (2) reporting on a process for developing a prototype tool for monitoring such an application (3) a discussion of future relevant directions for this and similar work

II. Methodology

We use an iterative process inspired by Design Thinking. Design Thinking is a hands-on, user-centric approach to problem-solving that deals with real users needs and consists of three overarching phases: Understand, Explore, and Materialize [2].

Understand and Explore. In the first two phases the researchers conduct user research to better clarify for whom the software artifacts are to be created and explore possible solutions. In our case, the user research consists of observation sessions of developers reverse engineering react applications followed by semi-structured interviews.

One important outcome of the initial user research is the definition of the end-user. Defining the end-user early in the process is essential to avoid designing solutions that do not address any user's real problem. In our case, we define the end user with the help of a user persona – a fictional yet realistic description of a typical user of an application. Having a User Persona helps the developer empathize with a specific end-user and prioritize features when developing the solution [3].

Materialize. In the third phase, we develop an initial prototype implementing a subset of the ideas generated in the initial phases. We develop the prototype iteratively with several evaluation sessions together with real end-users. After each evaluation, we prioritize the received feedback and implement the improvements. During the evaluation sessions, the end-users answer questions that help us determine what value the prototype can provide developers.

III. User Research

The user research for *react-bratus* consists of (1) an observation session of three students trying to reverse engineer the Zeeguu-React, the web UI for the zeeguu platform [7] and (2) three observation sessions with developers of different experience levels reverse engineering the Jira Clone application[3] followed by interviews. Table I summarizes the participants:

TABLE I
Interview + Observation Sessions

Type	**Person(s)**	**Developing Experience**	**React.js Experience**
Observation	Sara, Maja, Eleonora	Beginner	Beginner
Interview	Sara	Beginner	Beginner
Interview	Lukas	Experienced	Intermediate
Interview	Mads	Experienced	Experienced

[1]https://reactjs.org

[2]According to Stack Overflow Developers Survey 2020, ReactJS was voted as the most loved and wanted Javascript web framework.

[3]https://jira.ivorreic.com/project

A. Interviews and Observations

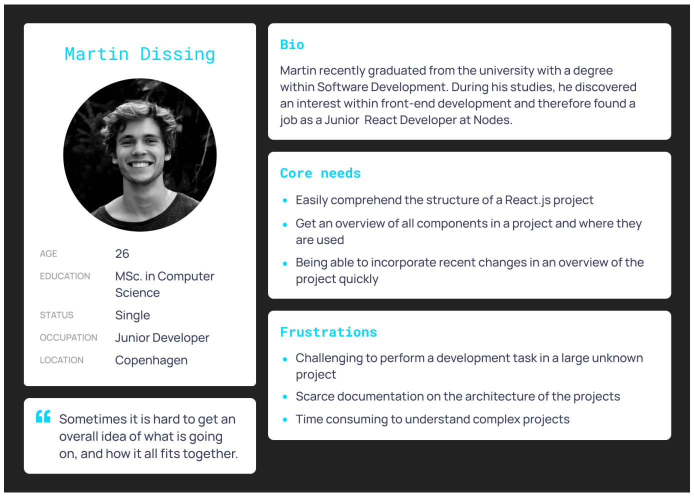
During the interviews, we discover that React.js projects rarely are documented, and reverse engineering an existing application is challenging regardless of experience level.

We observe that more experienced developers can reverse engineer applications faster. One possible reason is that with more experience comes more familiarity with their integrated development environments (IDE's) and available shortcuts. This enables developers to navigate through a repository and locate components of interest quicker. Experienced developers also spend less effort on understanding logic and can easier comprehend the component structure of an application.

At the opposite spectrum, junior developers spend more time familiarizing themselves with the file structure. In the observation study of the three beginners, we saw them explore the project as a team: one was navigating the code structure, all were discussing, one was drawing a component structure as their exploration proceeded. Moreover, junior developers spent more effort understanding syntax, logic and concepts from the React library while navigating a repository. Trying to understand both component structure and behaviour of an application simultaneously makes it more time consuming to comprehend an application for beginners.

B. User Persona

Based on the interviews and observations, we set the primary goal of *react-bratus* to support novice developers navigating existing React repositories and make these more understandable. Figure 1 presents our User Persona, Martin Dissing. We equip the user persona with personality traits such as name, age, education, place of living and a photo. While these details seem irrelevant to the solution, they help make the User Persona memorable [3].


Martin Dissing

AGE	26
EDUCATION	MSc. in Computer Science
STATUS	Single
OCCUPATION	Junior Developer
LOCATION	Copenhagen

“Sometimes it is hard to get an overall idea of what is going on, and how it all fits together.

Bio

Martin recently graduated from the university with a degree within Software Development. During his studies, he discovered an interest within front-end development and therefore found a job as a Junior React Developer at Nodes.

Core needs

- Easily comprehend the structure of a React.js project
- Get an overview of all components in a project and where they are used
- Being able to incorporate recent changes in an overview of the project quickly

Frustrations

- Challenging to perform a development task in a large unknown project
- Scarce documentation on the architecture of the projects
- Time consuming to understand complex projects

Fig. 1. User Persona

Martin recently graduated computer science student and started as a Junior Developer in a frontend bureau where he is working with React projects. He has limited experience with frontend development and is a beginner working with the React.js library. There is limited documentation on the project that he works on, and therefore, it is time-consuming to perform new development tasks. Therefore, his core needs are the following: The ability to quickly comprehend the structure of a React.js application, getting an overview of the components in a project and where they are rendered, and finally, being able to update the view to incorporate the latest changes quickly.

IV. React-bratus

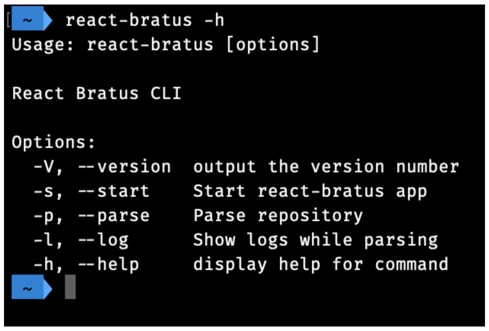
react-bratus is published as npm package and therefore installable by a react developer like any other dependency in the ecosystem by running `npm install`[4]. The source code is open-sourced on GitHub under an MIT license[5].

The implementation of *react-bratus* consists of 4 elements:

1) The CLI is responsible for accepting inputs from the user and can start the Server or parse a repository
2) The Web UI is responsible for displaying the extracted information to the user.
3) The Parser is responsible for traversing the source code and extracting information about a React.js application.
4) The Server is an express.js[6] application responsible for hosting the Web UI and provide the Web UI with data generated by the Parser through an Application Programming Interface (API).

A. The CL Interface

A user interacts with *react-bratus* either via the Command Line Interface (CLI) or via the web application in the browser. The CL interface opens the web application; this is inspired by other similar applications from the npm ecosystem[7].


```
 ~  react-bratus -h
Usage: react-bratus [options]

React Bratus CLI

Options:
  -V, --version  output the version number
  -s, --start    Start react-bratus app
  -p, --parse    Parse repository
  -l, --log      Show logs while parsing
  -h, --help     display help for command
 ~ 
```

Fig. 2. Command Line Interface

Figure 2 lists the available CLI commands. Executing `react-bratus --start` will launch a local web server and open the corresponding application that displays the component tree visualization. The component tree visualizes the rendered component hierarchy of React components starting from a root node.

[4]Package at https://www.npmjs.com/package/@react-bratus/cli

[5]https://github.com/stephanboersma/react-bratus All the references to the software in this paper refer to version v2.0.5 as can be found in the Releases section on GitHub or on npm

[6]https://expressjs.com

[7]e.g. Storybook (https://storybook.js.org) a popular UI component explorer

B. The Web UI

When opened for the first time, Web UI displays a legend which explains all the visual properties of the component tree. After closing this section, the main view is presented and consists of two parts: the component tree view and the sidebar (see Figure 3).

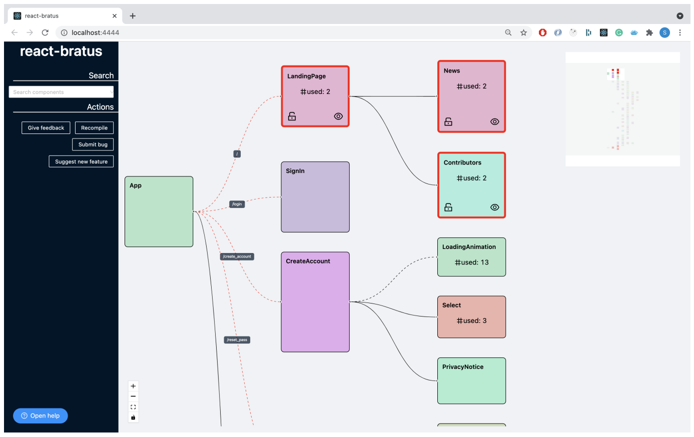


Fig. 3. The Web UI of *react-bratus* running on a dev machine

The component tree view contains control buttons, a minimap and the component tree. The control buttons allow the user to Zoom, centre the view and lock the components. The minimap provides a total overview to the user and displays where the user is looking in the visualization. Finally, the component tree consists of two elements: nodes and edges.

Each node in the visualization represents a user-defined React component. The height of each node is proportional to the lines of code in that component. Each node contains a label with the component name and a label indicating the number of times a component appears in the project. A hashing function based on the component name determines a unique background color for a node. This eases detecting reused components in the graph (e.g. Word and LoadingAnimation in Figure 4).

Each edge in the visualization indicates a "renders" relationship where the source node on the left-hand side renders the target node on the right-hand side. There are three types of relationships as illustrated also in Figure 4:

1) Always rendered subcomponents (solid black edges) – are always rendered by their parent (e.g. the TopTabs in Figure 4 are always rendered)
2) Conditionally rendered based on Javascript control flow (black dashed edges) – indicate a component rendered within a conditional block statement (e.g. the LoadingAnimation component from Figure 4 is only rendered if the main info is not available yet)
3) Conditionally rendered by the react-router-dom[8] (red dashed edges with labels) – rendered component given a specific path, represented as the label on the edge (e.g. since WordsForArticle and Starred are connected with red dotted lines the user knows that the two are alternative subpages of the WordsRouter component).

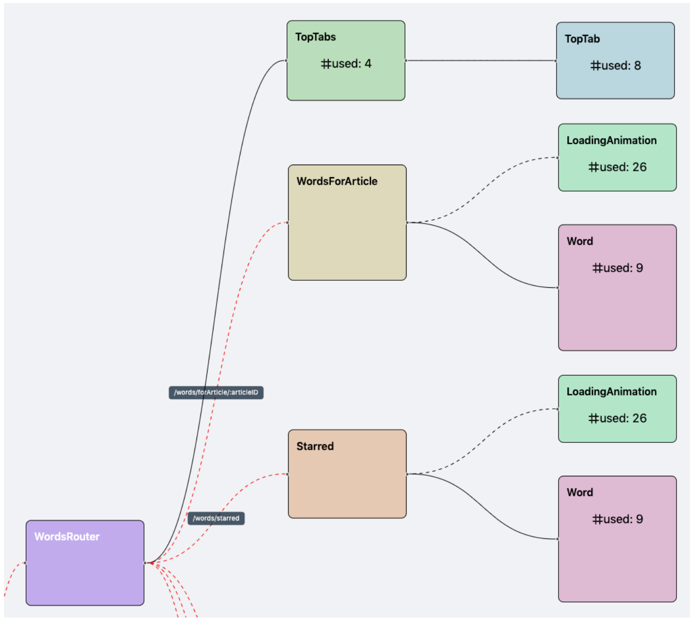


Fig. 4. The three types of edges: 1) solid = always rendered; 2) black-dotted = conditionally rendered (e.g. the Loading Animation component) and 3) red-dotted = conditionally routed by the react-router-dom package

A user can interact with a node in three ways: highlight, lock highlight and show details.

- The user highlights a node by hovering over the node with the mouse, and a thick red border and two icons will appear. Every occurrence of the highlighted node will appear with a red border in the view.
- Clicking the lock icon will lock the highlight and allow the user to navigate the visualization to find other component occurrences.
- The user can click on the eye icon to open a details panel for the given component. The component detail view displays the component's source code and the path link that opens Visual Studio Code [9] if installed.

The sidebar contains the search bar, actions buttons, and the help button. The search bar allows the user to search in the component tree hierarchy. When the user selects a search option, the component tree view will Zoom and pan to that component in the view and lock a highlight. The action buttons enable the user to submit feedback in the GitHub repository and recompile the project in case of changes.

C. The Parser

Our objective with parsing the source code is to identify all user-defined components and the other components they render. React.js is an extremely versatile library, and developers can apply its functionality in many ways. Parsing React.js and extracting components and their relationships is challenging for two reasons:

[8]The most popular routing package for React (https://reactrouter.com)

[9]https://code.visualstudio.com – the most popular JS IDE at the moment

1) React.js utilizes a Javascript syntax extension called Javascript XML (JSX). JSX allows developers to write HTML-like syntax in a Javascript file[10] and the two languages can be interleaved recursively.
2) React components are not first-class entities in the language and their extraction is complicated by the community's transition from an older standard (class-based components) to a newer one (functional components), the latter being much more popular because it is easier for developers to use; however, it is harder for static analysis to detect.

To parse Javascript and JSX, we utilize @babel/parser[11], a javascript parser that outputs an Abstract Syntax Tree (AST)[12] based on the ESTree specification[13]. We make use of the babel/traverse function to visit the AST and extract the necessary information.

Extracting Components. To extract components from code we observe that developers can define components with three kinds of AST nodes:

1) ClassDeclaration[14]
2) FunctionDeclaration
3) VariableDeclaration with an ArrowFunctionExpression

Based on this, the current version of the parser makes the following assumption: it identifies a component if and only if the component returns one or more JSXElements. This is a simplifying assumption, as it is perfectly possible for a component to not return a literal JSX expression but instead call a function that will eventually return one. Although in the experiments we conducted this was not a problem, it represents a current limitation of the parser.

Extracting Relationships. When the Parser finds a JSXElement tag, it checks to see whether that identifier corresponds to a component defined in the project. To simplify the Parser we assume that every component name is unique in a project. This assumption enables us to build a relationship based on the JSXElement identifier. If a project contains two separate components with the same name, *react-bratus* does not include them in the visualization and logs this info in the CLI output.

In React.js, it is possible to deal with routing in multiple ways. We only visualize routing if created with the react-router-dom package. React routes are highly customisable, enabling the developer to create routing logic, and we are therefore making the following assumptions to identify routing. We identify a routing relationship if and only if a component contains the word 'Route'. Then we look for the component rendered through that route in an attribute named either 'component' or 'render'.

[10]https://www.w3schools.com/react/react_jsx.asp

[11]https://babeljs.io/docs/en/babel-parser

[12]https://github.com/babel/babel/blob/main/packages/babel-parser/ast/spec.md

[13]https://github.com/estree/estree

[14]ClassDeclaration and later in the text, FunctionDeclaration, VariableDeclaration, ArrowFunctionExpression and JSXElements are AST type annotations. https://github.com/babel/babel/blob/main/packages/babel-parser/ast/spec.md

V. Outcomes of User Evaluations

This section presents the outcome of testing *react-bratus* with junior developers. There are two types of evaluations: written feedback and Zoom session. Written feedback is from two sessions, in which it was not possible to conduct testing through a virtual Zoom meeting.

Iteration 1. The first version of *react-bratus* contains limited features but the overall impression is positive. The developers describe the *react-bratus* CLI as (+) intuitive to interact with as a development tool. They find that (+) identifying components that have many lines of code is easy.

During the evaluation sessions, we discover a series of limitations that we address (our solution after "$\Rightarrow$"):

(-) It is difficult to locate all occurrences of a component in large projects $\Rightarrow$ we add support for highlighting all occurrences of a selected component on mouse-over

(-) There is no clear explanation about the visual properties used in the tool $\Rightarrow$ we implement the legend view

(-) *react-bratus* can not parse a project with several root components or projects with a root component that is not defined as 'App' $\Rightarrow$ a configuration file is introduced

Iteration 2. In this iteration, *react-bratus* is evaluated with three developers. (+) Developers get refactoring ideas by looking at the names, the size and the children of components. (+) The help view containing the legend receives positive feedback (e.g., *"I really like seeing this help section at startup. Let us be honest, developers are very lazy. No one wants to read your boring documentation on GitHub" (Emil)*.

We also address several limitations:

(-) Three developers use the in-browser search to find a specific component (and emphasize this as an essential feature); however the in-browser search is successful only if the component of interest is visible $\Rightarrow$ we implement system-wide search

(-) The available interactions with components are not apparent $\Rightarrow$ we implement icons for the clickable interactions and more conspicuous change of color when the state changes.

(-) A developer suggests adding color mapping to the nodes help identify recurring components $\Rightarrow$ we implement the hash value of a component name to colour the node inspired by [6].

(-) A user attempts to click on a component in the visualization to see additional information, but nothing happens; the developer wants to know in which file the component is defined $\Rightarrow$ the detail view is added with the file view and the link to opening Visual Studio straight from the browser

Iteration 3. In the final iteration, *react-bratus* is evaluated by three junior developers. (+) The developers emphasise that the search feature is useful in locating components. (+) The component detail view is well received because it allows the developers to see the code of the component or even navigate to the IDE. We still discover limitations that we address:

(-) the parser crashes if a circular dependency exists in the source code $\Rightarrow$ we increase robustness of parsing and add a warning message in the presence of a circular dependency

(-) trivial components are rendered relatively larger, and thus, more important than they are in reality due to the minimal size required to render the name and usage count for a component ⇒ we add the option of mapping the size of a component on the color intensity of a node with a monochromatic scale; reinforcing the size information allows now to better distinguish trivial components

VI. Discussion

There are still several unsolved limitations of *react-bratus*:

(-) Focusing on a subset of the component tree; in the case of large applications, the whole tree can become hard to navigate; being able to hide everything but the subtree of interest would probably be required in such situations

(-) More React-specific information can enrich the semantics of the visualization: e.g. the properties for a component, the state variables, the proportion of HTML and JS code in a component, etc. Extracting this information and finding the best way to visualize them is still a challenge

(-) Statically parsing React.js source code and extracting components with 100% precision is a challenging task

One limitation of our work is that the number of junior developers we interacted with was small and they might not be representative of the whole junior developer population. Their feedback might have also been biased by their desire to be polite to the authors of the tool. Moreover, since in our user research we only interviewed one expert, we might have concluded too early that the tool would be better fit for juniors. It would still be important to see whether and in which way a tool like Bratus can be valuable for senior developers. Finally, also educators could benefit in scenarios where they have to evaluate projects developed by their students.

Another big limitation of the work is that the tool was used in a once-off scenario, not integrated in the daily routine.

VII. Related Work

React-Sight[15], React-ScopeReact-Scope[16], and Realize[17] are developed as Google Chrome extensions. When a user opens the development tools tab in Google Chrome, they inject scripts that collect data about a React.js application. The tools present the user with information about the component hierarchy and the state of each component. React-Sight and Realize provide interactions such as searching, filtering, highlighting and displaying information about a specific component. React-Scope only enables the user to see the component tree and inspect the state of each component.

React-Monocle[18] is a tool that provides the component hierarchy differently than the other tools. Developers install React-Monocle as a CLI and initiate the tool from their terminal. React-Monocle parses through the source code to generate the component hierarchy and launches a web application displaying the visualization and the React.js application side by side. When the application changes, showing more elements than initially, the visualization automatically updates to represent the current state. The developers achieve this by injecting wrapper functions around all 'setState' calls that are responsible for updating the visualization.

Compodoc (https://compodoc.app) is an automatic documentation tool developed for Angular. It provides an overview of dependencies between components, services, modules, and providers within an Angular application. In large applications, this view can become a complex and messy.

Polymetric views is a lightweight visualization technique incorporating software metrics information in the position, width, height, and colour of nodes [4]. Although we take inspiration from the approach, we only use the height and color of a node to convey information.

Streamsight is a visualization tool that helps developers inspect, monitor, and comprehend the dynamic behaviour of streaming applications [6]. It presents processing elements represented as nodes in a graph with their respective inputs and outputs represented as edges. Just like we do, nodes are colored based on string attributes of the processing element by using a hashing function to map the attribute to a colour.

VIII. Conclusions and Future Work

We presented the development of *react-bratus*, a tool that aims to support junior developers in understanding the component structure of React.js applications. We conducted user research to define our typical end-user, a novice React.js developer. Throughout three iterations, we described how *react-bratus* evolved, leveraging feedback from six developers who tested the prototype in different stages of development. In testing, we discovered that visualising the rendered component hierarchy promises to be helpful for novice developers. However, this is only the beginning; further tests and development and more in-depth evaluations are required to further understand how such a tool can best support developers.

References

[1] Merino, L., Ghafari, M. and Nierstrasz, O., 2016. Towards actionable visualization for software developers. Journal of Software: Evolution and Process, 30(2), p.e1923.

[2] Gibbons, S., 2016, "Design Thinking 101," URL: https://www.nngroup.com/articles/design-thinking/

[3] Harley, A., 2015 "Personas make users memorable for product team members," URL: https://www.nngroup.com/articles/persona/

[4] Lanza, M. and Ducasse, S. 2003 "Polymetric views – a lightweight visual approach to reverse engineering," IEEE Trans. Softw. Eng., vol. 29, no. 9, pp. 782–795.

[5] Slater, J., Anslow, C., Dietrich, J. and Merino, L., 2019, "CorpusVis – Visualizing Software Metrics at Scale," Working Conference on Software Visualization (VISSOFT)

[6] De Pauw, W. and Andrade, H., 2009, "Visualizing Large-Scale Streaming Applications," Information Visualization, 8(2), pp.87-106.

[7] M.F. Lungu, L. van den Brand, D. Chirtoaca, M. Avagyan, 2009, "As We May Study: Towards the Web as a Personalized Language Textbook," Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems

[15]https://github.com/React-Sight/React-Sight

[16]https://github.com/React-Scope/React-Scope

[17]https://github.com/oslabs-beta/Realize

[18]https://github.com/team-gryff/react-monocle