From GWT to Angular: An Experiment Report on Migrating a Legacy Web Application

Benoît Verhaeghe, Anas Shatnawi, Abderrahmane Seriai, Anne Etien, Nicolas Anquetil, Mustapha Derras, Stephane Ducasse

Department: Head
Editor: Name, xxxx@email

From GWT to Angular: An Experiment Report on Migrating a Legacy Web Application

B. Verhaeghe
Berger-Levrault, France
Université de Lille, CNRS, Inria, Centrale Lille, UMR 9189 – CRIStAL, France

A. Shatnawi
Berger-Levrault, France

A. Seriai
Berger-Levrault, France

A. Etien
Université de Lille, CNRS, Inria, Centrale Lille, UMR 9189 – CRIStAL, France

N. Anquetil
Université de Lille, CNRS, Inria, Centrale Lille, UMR 9189 – CRIStAL, France

M. Derras
Berger-Levrault, France

S. Ducasse
Université de Lille, CNRS, Inria, Centrale Lille, UMR 9189 – CRIStAL, France

***Abstract*—Berger-Levrault is an international company that developed applications in GWT for more than 10 years. However, GWT is no longer actively maintained, with only one major update since 2015. To avoid being stuck with legacy technology, the company decided to migrate its applications to Angular. However, because of the size of the applications (more than 500 web pages per application), rewriting from scratch is not desirable. To ease the migration, we designed a semi-automated migration approach that helps developers migrate applications' front-end from GWT to Angular and a tool that performs the migration. In this paper, we present our approach and tool. We validated the approach on concrete application migration and compared its benefits to redeveloping the application manually. We report that the semi-automated migration offers an effort reduction over a manual migration. Finally, we present recommendations for future migration projects.**

COMPANIES use GUI frameworks to ease the creation of their applications' front-end. However, those frameworks are getting old and become legacy. In such a situation, companies must migrate their applications to more recent frameworks to avoid being stuck with old technologies.

Berger-Levrault has developed several applications using the Google Web Toolkit (GWT) framework. This framework allows one to write the front-end of a web application in Java. However, GWT received only one major update since 2015. Thus, the company decided to migrate its applications to Angular.

Berger-Levrault evaluated the time needed to redevelop one of its GWT application manually at 8,000 person-days. It includes the migration of more than 500 web pages written with several million lines of code. Considering the size and the number of applications (eight applications targeted for GWT to Angular migration), it is clear that manual migration is not feasible in this industrial context.

To help the company migrate its projects, we designed an iterative semi-automated migration approach. This approach includes a tool called Casino described in a previous paper [1].

In this paper, we first present the migration context and the architecture of the applications to be migrated. Then, we detail the input and output of each step of our approach and how developers can fine-tune Casino along the process for the benefit of the next iteration. We also discuss which actions can be reused and the ones that must be performed for each migration project. Finally, we present a concrete migration experiment. We give figures such as the number of migrated UI elements, time spent, *etc.*, and compare our approach results with a manual migration.

Migration Context

In the following, we detail the existing migration approaches and the architecture of the applications at Berger-Levrault.

Modernization approaches

Many migration approaches were proposed for the modernization of applications [2], [3], [4], [5]. They can be divided into three categories [6]: from scratch, wrapping, and semi-automated migration.

From scratch consists of recreating the full application manually. This solution is adopted for small applications but can not be used for big applications as in the Berger-Levrault context.

Wrapping consists in executing the old application in a new context [7]. Although wrapping provides a fast way to upgrade applications for the end-user, it is a temporary solution for the developers rather than a concrete migration.

Semi-automated migration consists of using tools to migrate completely or partially the application. In the case of partial migration, developers only need to deal with the complex code that can not be automatically migrated. In the end, the application is fully written in the target language/framework.

Another approach, transpilers, migrate from one programming language to another without considering GUI frameworks' specificities [8]. For example, JSweet [9], migrates Java to TypeScript without focusing on GUI and therefore produces poor results in this context.

The semi-automated approach is the most adopted in literature [2], [3], [10], [11], [12], [13]. However, the authors focused on the conception of tools that perform GUI migration and fail to detail the migration process and how their tools fit in. The papers we found are often theoretical, presenting the technical solutions but lacking a practical evaluation of how well developers of a legacy application could perform the kind of incremental migration that is required on a large application. Our partner also needed a better estimation of the investment required to perform a migration.

Application architecture

To perform a semi-automated GUI migration, we first need to identify the architecture of the applications. Both GWT and Angular allow developers to write applications that follow the Single Page Application (SPA) style.

This high-level architecture is preserved during the migration. However, there are technical differences between GWT and Angular that must be taken into account.

Figure 1 presents an example of a generic web application architecture. It is divided into two

Published by the IEEE Computer Society

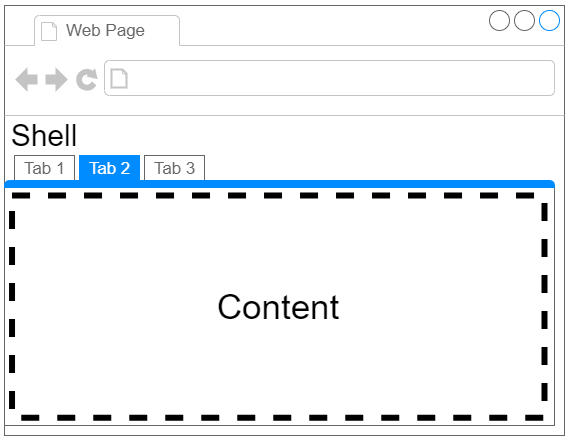

Figure 1: Main parts of a web application architecture

parts: *Shell* [14] and *Content*.

The **Shell** corresponds to the infrastructure provided by the GUI framework. It includes the company specific rules and configuration. It also includes the front-end header and footer of the web pages.

The **Content** part is developed for each application. It corresponds to the web page GUI and the business code (*e.g.* the application rules, distant server address, application-specific data).

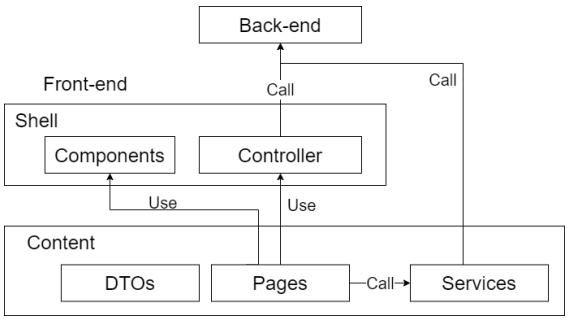

Figure 2: Architecture of web applications

Figure 2 details the Shell and the Content.

The *Shell* part is subdivided into the *components* and the *controller*.

The **Components** (also known as widgets) are the UI elements that can be used by developers to design their applications.

The **Controller** renders the header and footer of the application, ensures the security layer of the front-end (*i.e.*, inject the session ID, redirect to the login page...) and, manages URL mapping (*i.e.*, display the page selected by the user).

The *Content* part of the architecture is divided into the *Data Transfer Objects* (DTOs), the *pages*, and the *services*.

The **DTOs** are the data manipulated by the application. In the object-oriented paradigm, they correspond to classes with attributes. Attributes type can be: primitives (*i.e.* string, int *etc.*), collections, dictionaries, and other DTOs.

The **Pages** are the GUI and behavioral code (*e.g.*, the code executed when the user interacts with a widget) designed by developers and used by customers. They are designed using *components* of the Shell, and interact with other pages through the *controller*. In particular, they call the *controller* for navigation between pages and transmitting data to other pages. Some *pages* uses *DTOs*. It is the case for pages displaying data retrieved from the back-end. Those *DTOs* are provided by *services*.

The **Services** are the connection between the front-end and the back-end of the application. When called, they create a request to the back-end, receive the result, and transmit the result to the original caller (*i.e.*, a page or another service).

Table 1: Application architecture technical implementation

	GWT	Angular
Components	Java file	TypeScript, HTML, and CSS files
Controller	Java and XML files	Module and route file
DTOs	Java class	TypeScript class without getter and setter
Pages	Java file	TypeScript, HTML, and CSS files
Services	GWT/RPC	REST

Table 1 summarizes the main technical implementation differences between GWT and Angular. In general, GWT uses Java files, and Angular uses TypeScript, HTML, and CSS files. There is also a major difference in the services. GWT uses a specific GWT/RPC protocol, whereas Angular uses REST.

We will now look at a migration process that will ease the transition from one architecture to another.

Migration process

To perform the migration from GWT to Angular, we designed a process divided into three

steps: shell migration, back-end connection migration, and front-end migration.

Shell migration

The first step to migrate the applications' GUI is the migration of the Shell in the target framework. This step consists of manually developing in the target framework the pages' header and footer, the CSS, and the company's widgets.

Since the Shell is the same for all the applications, this step needs to be done only once for all migration projects.

Once the Shell is migrated, it is possible to perform the migration of the back-end connection and the migration of the front-end.

Back-end connection migration

To perform the migration of the back-end connection, we used the Casino tool (see Figure 3).

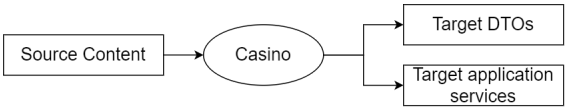

Figure 3: back-end connection migration

Casino is an engine that translates GWT GUI to Angular GUI [1]. It performs a static analysis of the GWT source code to extract the front-end of the application (*i.e.*, the widgets, their attributes, and the widgets composition), its behavior, its services, and its DTOs. Internally, Casino uses a mapping of the GWT widgets to their Angular counterparts. This mapping might not be exhaustive, and new widgets can be incrementally added when they are first encountered.

At this step, Casino creates the services in Angular. It transforms the JAVA RPC endpoints (*e.g.*, methods) into Java REST endpoints. Note, the Java REST endpoints, in our case, do not follow RESTful standards, which makes it easier to migrate them (*e.g.*, we do not access element state, but we call methods). Casino also generates the Angular services to request the Java REST endpoints. For the DTOs, Casino extracts their structures and creates their Angular counterparts.

During this step, all the services and DTOs of the application are migrated automatically by Casino. Thus, it only needs to be executed once for each application migration.

Front-end migration

Finally, it is possible to migrate the front-end of the application. Figure 4 presents the migration approach. It consists of three main steps that are repeated over all pages to migrate.

Select a Page in the source application. For example, developers can start with simpler pages or pages that only use widgets already known by Casino. They can also select to migrate a group of pages that work together to avoid dangling dependency issues.

Use Casino to analyze the selected page(s) and generate it(them) into the target framework. This step is automatic.

Fix the page presenting differences between their source and the target versions. Differences are identified "manually", they include visual differences (*e.g.* widgets are missing or the page layout is incorrect), and behavioral differences (*e.g.* the proper behavior is not executed when the user interacts with the page). If there are differences, developers fix them and integrate the migrated page in the final application. Some fixes (for example, adding new widgets to the mapping) can be retro-fitted into Casino to improve the next page migration.

Evaluation

Case study

We evaluate our semi-automated approach on a GWT application of Berger-Levrault. This application is called Omaje and was selected as representative of other Berger-Levrault's applications. Omaje is a client subscription management

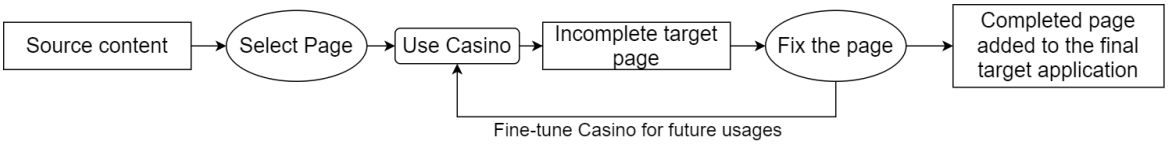

Figure 4: Front-end migration

application used internally., It is, therefore, a safe case study for our experiment. Omaje includes 20 main pages following the SPA architecture and contains 6,683 GWT elements. It is built with 33 different kinds of widgets, from basic ones (button), to complex ones (charts or tables that auto-update part of the GUI when a row is selected). In total, in its original version, Omaje "weighs" 191 KLOC that are implemented using 2,669 classes and 14,882 methods.

We hired a Master student as a trainee to perform the migration of the GUI of Omaje from GWT to Angular. We wanted somebody who did not know Angular yet (representative of many developers in the company) nor our tool. The drawback of this choice is that the student does not know Omaje either, which would not be the case with a company developer.

The trainee required 10 person-days to install the application environment, discover the Angular framework, and learn how to use our migration approach and tool. Then, he performed the front-end migration. It consisted of following the process described above (selecting a page, migrating it with Casino, fixing and integrating it into the Angular application). When fixing a page, he encountered GUI elements not migrated by Casino because they only exist in the source framework. In this case, he created a corresponding component in the target framework and added it into the Casino's widget map.

Effort reduction

Once the environment was installed, the GUI migration was completed in 14 days.

The developers of Omaje had roughly estimated the effort to manually migrate the application to 104 person-days. Although this is a very crude estimation, there is no doubt that our tool allowed the trainee to achieve the same result in considerably less time, and with the added disadvantage that he did not know the application itself at the beginning.

In total, the migration cost 24 days. The migrated application consists of 505 Angular files and 34,830 LOC. It is 82% less than the original application. The main reason is that TypeScript and HTML are less verbose than Java.

Reduction of manual work along the process

As we mentioned earlier, when the migration developer identified a missing GUI element in the Angular application, he developed a generic reusable Angular component. This component is then used by Casino during the generation of other pages.

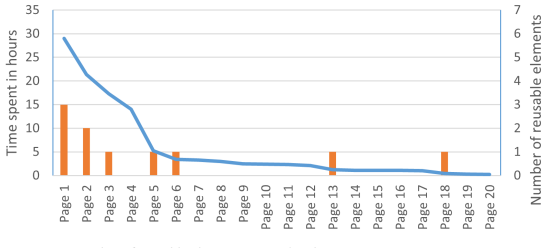

Figure 5: Results of manual work reduction gained by reusable components to extend Casino tool

■ Time spent to manually complete the migration and create reusable Angular elements in hours (left axis)
■ Number of reusable elements created in this page (right axis)

Figure 5 presents the amount of time in hours required by the developer to perform the migration of each page, with the number of reusable Angular elements created for those pages. The results show that 10 reusable Angular GUI elements have been developed during the migration. One can see that the time steadily decreased as the trainee gained experience with the tool and the required widgets were added.

The time needed to complete the migration of pages 13 and 18 is low despite the creation of two new components. The two components were really simple (*i.e.*, consist of less than three HTML tags), and their behavioral logic was already existing. Thus, nearly no time was required to create them.

We note that the main effort was to add missing Angular elements in the application. Once this step was done, migrating a page became trivial for the Master student. Setting up an expert team dedicated to designing the target Angular elements would have eased the migration.

Maintainability of the produced application

A common problem with automatically generated code is the quality of the code. We evaluate this aspect using the SonarQube engine that

provides code quality information over a project. We focused on three aspects: reliability, maintainability, and security. Reliability validates the absence of potential bugs. Maintainability checks that the migrated application will be usable by developers after the migration. Security verifies that the migrated application does not contain vulnerabilities.

Table 2: SonarQube number of issues per quality aspect

	Reliability issue	Maintainability issue	Security issue
Original	1,389 (1%)	5,075 (3%)	4 (0%)
Semi-automatic	684 (1%)	1,096 (5%)	1 (0%)
Full migration	409 (1%)	911 (3%)	0 (0%)

In parentheses: percentage of issues per line of code

Table 2 summarizes the result for the code quality evaluation. It presents the number of issues reported for each aspect. It shows that our semi-automatic migration tool did not damage the quality of the code. Although the percentage of issues per line of code is constant, the number of issues has decreased in the migrated application. In fact, by standardizing the former code into the target standard, Casino allows one to remove many Sonar issues. Such standardization of the code is similar to the one performed by [15] to ease the migration process. This is also a common practice when improving an application's maintainability.

The Omaje development team later performed 56 functional test scenarios on the Angular version and did not report any bug. The team also developed new features in the migrated Angular application and did not report any problem. They decided to adopt this version for future development.

Conclusion

In this paper, we expose a concrete problem of GUI migration. We presented an approach to migrate the front-end of applications and applied it to an application of Berger-Levrault. We report that our approach allows the company to reduce the migration time significantly.

In our experiment, the manual step of the migration has been performed by only one developer. Future work includes the migration of larger applications with a larger team.

As final recommendations, we stress the benefits of standardization of the source code following coding conventions to ease the migration process. This reduces the manual effort to discover and map source widgets to their target counterparts, and it improves the maintainability of the code.

We also encourage future practitioners to pay attention to the differences between the source and target GUI framework architecture style. In our example, both used the SPA architecture style, which did not require additional work. In case the two GUI frameworks are based on different architectures, one has to define rules to migrate from one architecture to the other.
We provide at https://badetitou.github.io/projects/Casino/ links to several GUI importers and generators that can help migrate applications and provides details on our approach.

REFERENCES

1. B. Verhaeghe, A. Etien, N. Anquetil, A. Seriai, L. Deruelle, S. Ducasse, and M. Derras, "GUI migration using MDE from GWT to Angular 6: An industrial case," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER'19)*, Hangzhou, China, 2019, pp. 579–583. [Online]. Available: https://hal.inria.fr/hal-02019015
2. K. Garcés, R. Casallas, C. Álvarez, E. Sandoval, A. Salamanca, F. Viera, F. Melo, and J. M. Soto, "White-box modernization of legacy applications: The oracle forms case study," *Computer Standards & Interfaces*, pp. 110–122, Oct. 2017.
3. O. Sánchez Ramón, J. Sánchez Cuadrado, and J. García Molina, "Model-driven reverse engineering of legacy graphical user interfaces," *Automated Software Engineering*, vol. 21, no. 2, pp. 147–186, 2014. [Online]. Available: http://link.springer.com/10.1007/s10515-013-0130-2
4. H. Samir, A. Kamel, and E. Stroulia, "Swing2script: Migration of Java-Swing applications to Ajax Web applications," in *14th Working Conference on Reverse Engineering (WCRE 2007)*, 2007.
5. E. Shah and E. Tilevich, "Reverse-engineering user interfaces to facilitate porting to and across mobile devices and platforms," in *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, \& VMIL'11*. ACM, 2011, pp. 255–260.

6. H. M. Sneed and C. Verhoef, “Cost-driven software migration: An experience report,” *Journal of Software: Evolution and Process*, p. e2236, 2020.
7. T. Tonelli *et al.*, “Swing to swt and back: Patterns for api migration by wrapping,” in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–10.
8. J. Brant, D. Roberts, B. Plendl, and J. Prince, “Extreme maintenance: Transforming Delphi into C#,” in *ICSM’10*, 2010.
9. R. Pawlak, “Jsweet: Insights on motivations and design,” *A transpiler from Java to JavaScript. EASYTRUST*, vol. 16, 2015.
10. T. Hayakawa, S. Hasegawa, S. Yoshika, and T. Hikita, “Maintaining web applications by translating among different RIA technologies,” *GSTF Journal on Computing*, p. 7, 2012.
11. F. Fleurey, E. Breton, B. Baudry, A. Nicolas, and J.-M. Jezéquel, “Model-Driven Engineering for Software Migration in a Large Industrial Context,” in *Model Driven Engineering Languages and Systems*, G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, Eds., vol. 4735. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 482–497. [Online]. Available: http://link.springer.com/10.1007/978-3-540-75209-7_33
12. A. Mesbah and A. van Deursen, “Migrating multi-page web applications to single-page ajax interfaces,” in *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, ser. CSMR ’07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 181–190. [Online]. Available: http://dx.doi.org/10.1109/CSMR.2007.33
13. S. Bragagnolo, N. Anquetil, S. Ducasse, S. Abderrahmane, and M. Derras, “Analysing microsoft access projects: Building a model in a partially observable domain,” in *International Conference on Software and Systems Reuse (ICSR’20)*, ser. LNCS, no. 12541, Dec. 2020.
14. A. Osmani, “The app shell model,” https://developers.google.com/web/fundamentals/architecture/app-shell, accessed: 2020-09-10.
15. L. Włodarski, B. Pereira, I. Povazan, J. Fabry, and V. Zaytsev, “Qualify first! a large scale modernisation report,” in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 569–573.

Benoît Verhaeghe is a Ph.D. student at the RMoD Team of Inria Lille - Nord Europe, Lille, France, and research engineer at Berger-Levrault, France. His research interest includes reverse engineering, maintenance, and software systems migration. Contact him at benoit.verhaeghe@berger-levrault.com.

Anas Shatnawi is a research engineer at Berger-Levrault, France. He obtained his Ph.D. degree in Computer Science from LIRMM of University of Montpellier, France. His research interest is in software reuse, reengineering, reverse engineering and empirical software engineering. Contact him at anas.shatnawi@berger-levrault.com.

Anne Etien is full Professor at the University of Lille, France. Her research interests concern the reengineering of complex legacy systems, tests, software migration. Contact her at anne.etien@inria.fr.

Nicolas Anquetil is assistant Professor at the University of Lille, France. His research interests include everything related to software evolution, including: software reverse engineering, software quality, tests, software migration. Contact him at nicolas.anquetil@inria.fr.

Abderrahmane Seriai obtained his Ph.D. in computer science from the University South Brittany (France) in 2015. He joined the Berger-Levrault group in 2017. His current work revolves around applied research in the field of software engineering. He actively participates in the scientific research, design, analysis and development aspects of use cases for new technologies (software migration, product lines, big data, blockchain, etc.). abderrahmane.seriai@berger-levrault.com.

Mustapha Derras is research director at Berger-Levrault. Contact him at mustapha.derras@berger-levrault.com.

Stephane Ducasse is research director at INRIA Lille leading the RMoD Team, France. During 10 years, he co-directed with O. Nierstrasz the Software Composition Group. He is president of ESUG. He is one of the leaders of Pharo: a new exciting dynamic language. Contact him at stephane.ducasse@inria.fr.