

A Hybrid Architecture for the Incremental Migration of a Web Front-end

Benoît Verhaeghe^{1,2}^a, Anas Shatnawi¹^b, Abderrahmane Seriai¹, Anne Etien²,
Nicolas Anquetil²^c, Mustapha Derras¹ and Stéphane Ducasse³

¹*Berger-Levrault, France*

²*Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 – CRISTAL, France*

³*Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 - CRISTAL, France*

Keywords: Software Transformation, Incremental Migration, Hybrid Architecture, GWT, Angular.


Abstract: Nowadays, software migration is an effective solution to adopt new technologies while reusing the business value of existing applications. Among other challenges, the size and complexity of large applications are obstacles that increase the risks of migration projects. Moreover, the migration can imply a switch of programming languages. This is the case when migrating from Java to TypeScript. Thus, it is hard to migrate large and complex applications in one straightforward step. Incremental approaches have been designed to counter this problem. These approaches are based on hybrid architecture usages. However, none of the approaches use a hybrid architecture for GUI defined with different programming languages. In this paper, we propose a new hybrid architecture that enables the incremental migration of web applications. Our architecture is based on Web Components that allow legacy technology artifacts to work with modern ones. We implement the architecture and use it in the case of migrating GWT applications to Angular. Then, we validate its usability in a real context by migrating an industrial web application.


1 INTRODUCTION


In the context of a collaboration with Berger-Levrault, a major IT company, we work on the migration of web applications written in GWT (Google Web Toolkit) to Angular.

Approaches easing the migration of such a front-end application have already been proposed (Robillard and Kutschera, 2019, Sánchez Ramón et al., 2014, Verhaeghe et al., 2021). They propose a partial GUI migration, manually fixing the remaining code, and delivering the new system. However, the manual step can require many man-months of effort for large applications. Thus, the final user would not receive any update during this period and developers would be unable to develop new functionalities in the legacy technology (planned to be abandoned) or the new one (not yet up and running). Such an approach is not feasible in an industrial context with a strong pressure to deliver.

One solution is to migrate the application incrementally. Kontogiannis et al. (2010), and Teppe (2009) proposed to migrate a part of an application, integrating it into a hybrid architecture mixing legacy and migrated parts, and delivering the hybrid application to the final user. However, their hybrid architecture can not be adapted straightforwardly to the web GUI context. Indeed, communication between the legacy and migrated parts is based on a foreign function interface (FFI) (Polito et al., 2020) that enables one programming language to call methods defined in shared libraries. FFI is a common solution for desktop applications. However, one can not use it for web applications. Indeed, a website accessing user personal files would create a security threat. Robillard and Kutschera (2019) used a hybrid architecture to mix JavaSwing and JavaFX, but their solution is simplified by the use of the same language (Java) in both frameworks. In our case, GWT uses Java while Angular uses Typescript. Moreover, their hybrid architecture is based on an already existing JavaFX feature allowing one to integrate JavaSwing components inside a JavaFX GUI. Such a feature is not available for several GUI frameworks.

^a  <https://orcid.org/0000-0002-4588-2698>

^b  <https://orcid.org/0000-0002-5561-4232>

^c  <https://orcid.org/0000-0003-1486-8399>

To support the migration of large industrial web GUI applications, we designed a hybrid architecture that authorizes one to mix GUI defined with different GUI frameworks using different programming languages. It also allows communication between the different GUIs.

We implemented the hybrid architecture and used it during an industrial migration project from GWT to Angular. The results show that our hybrid architecture enables the migration of the application. It allows one to mix GWT and Angular without perceivable performance issues.

The contributions of this paper are:

1. a hybrid architecture that allows one to integrate legacy and modern modules in the same web application;
2. an implementation of our hybrid architecture for the case of migrating GWT applications to Angular ones; and
3. an evaluation of the usability of the hybrid architecture on an industrial application.

The remaining of the paper is organized as follows: In Section 2, we review the literature on hybrid architectures. In Section 3, we present our hybrid architecture. In Section 4, we present the implementation of our architecture in the case of mixing GWT and Angular. In Section 5, we evaluate our hybrid architecture by migrating part of an industrial GWT application to Angular pages. In Section 6, we present threats to validity. In Section 7, we conclude and present future work.

2 RELATED WORK

To perform an incremental GUI migration, the solution proposed in the literature is to use a hybrid application mixing both the source and target GUI (Kontogiannis et al., 2010, Robillard and Kutschera, 2019, Teppe, 2009). We identified various publications related to designing or using a hybrid architecture.

First, authors report several issues that must be considered when designing a hybrid architecture. Terekhov and Verhoef (2000) and Chisnall (2013) detail the challenges to make two programming languages interoperable and Robillard and Kutschera (2019) present the challenge raised when mixing GUIs:

Communication. In the case of a hybrid application, several programming languages might be involved in the implementation. Each language has to communicate with the other(s) (*e.g.*, invoking

methods from one programming language to another). Chisnall (2013) details the difficulties of bridging two programming languages, reporting the need for C interfaces to enable communication between Java and C++.

Type Matching. Another major challenge is the matching of data types (Chisnall, 2013, Terekhov and Verhoef, 2000). The two programming languages might have different structure representations for a type. For instance, Java primitive types are implemented using specific Java wrapping classes, whereas, in JavaScript, primitive types are classic JavaScript types. Thus, a *number* in JavaScript can not be directly translated into a Java *Integer*.

GUI Mixing. The need to mix GUI has only been raised by Robillard and Kutschera (2019). In their context, widgets defined in different GUI frameworks are present within the same page. Thus, a strategy must be developed to enable the integration of one GUI with the other.

Then, we identify existing research projects using a hybrid architecture and the issues they deal with.

Robillard and Kutschera (2019) work on the migration of a Java Swing application to JavaFX. They migrated the application incrementally by mixing Java Swing and JavaFX components. Because both framework uses Java, the *communication* and *type matching* issues do not arise. The *GUI mixing* issue is overcome using interoperability features of JavaFX¹.

Comella-Dorda et al. (2000) detail different strategies to mix application. To mix user interfaces, they propose to use a screen scraping technique. It consists of analyzing the source application rendered UI at runtime, converting it, and wrapping it for the target platform (web-based or desktop-based). Flores-Ruiz et al. (2018) and Zhang et al. (2008) use this strategy with different implementations. Although this approach allows one to present the GUI of an application in another context (*e.g.* desktop-based GUI inside a web browser), it does not discuss how enabling communication between the source GUI and the target one. Using screen scraping technique, the authors deal with the GUI mixing constraint but do not consider the *communication* and the *type matching* between the hybridized elements.

Kontogiannis et al. (2010) propose a set of transformation rules to migrate from one programming language to another. Whereas they do not discuss how two different languages can communicate, their transformation rules deal with the *type matching* problem.

¹<https://docs.oracle.com/javase/8/javafx/interoperability-tutorial/swing-fx-interoperability.htm>

Table 1: Hybrid architectures fulfilling issues.

	Communication	Type matching	GUI mixing
Robillard and Kutschera (2019)	✓		✓
Comella-Dorda et al. (2000)			✓
Flores-Ruiz et al. (2018)			✓
Zhang et al. (2008)			✓
Kontogiannis et al. (2010)		✓	
Teppe (2009)		✓	
Sneed et al. (2006)	✓	✓	
<i>Technical: iframe</i>			✓

To do so, they created a type correspondence table between PL/IX and C. *GUI mixing* was out of their scope.

Teppe (2009) uses an iterative approach to migrate an SPL² application to C++. When performing the migration, he had to deal with the *type matching* problem. The author uses a type correspondence table and an intermediate layer that transforms a type defined in one language to its equivalent type in another language. This approach is only detailed for applications without GUI and running on the desktop (rather than the browser).

Finally, Sneed et al. (2006) propose to wrap legacy code to make it available as a web service. This strategy allows one to create *communication* between different programming languages. Relying on web services, they also deal with the *type matching* problem by serializing data in XML format.

Additionally, a common way to mix web GUI in industry is the usage of the *iframe* tag³. It allows one to insert inside one web page the content of another one. This is convenient as it does not require designing a new architecture. However, it comes with substantial limitations as it is strongly discouraged to enable communication with the content of an *iframe* for security purposes, and the *iframe* content should not access the main page.

Table 1 summarizes the existing migration solutions, none deal with the three problems identified. Thus, to enable the incremental migration of large applications, one needs to consider all these issues and propose solutions.

3 HYBRID ARCHITECTURE

Our hybrid architecture allows one to mix, inside a web application, two GUIs defined with two different web frameworks, possibly in different programming

languages. In the following, we present the architecture we designed to mix two GUIs and how it is used to build an application.

Section 3.1 presents the hybrid architecture. Section 3.2 details the usage of Web Components to mix GUIs in our hybrid architecture.

3.1 Hybrid Architecture Description

Our hybrid architecture tackles the three issues depicted in the literature (Section 2): *communication*, *type matching*, and *GUI mixing*.

Figure 1 presents the hybrid architecture in which the front-end is divided into three parts: controller, original pages (not migrated), and migrated pages.

The controller is the central part of our hybrid architecture. It can be developed in any programming language, independent of the source or target GUI frameworks. It is the front-end entry point of the web application. End-users access web pages through the controller by requesting URLs. The controller also manages the context of the application (*e.g.*, the currently logged user, the page currently displayed, *etc.*). Finally, it is in charge of rendering web pages header and footer.

The pages (original and migrated) contain the actual application GUI. No direct communication between pages is allowed, navigation and data transfer must go through the controller. For instance, a page can not request the value of a text field from another page. Pages are migrated one at a time, entirely, *i.e.*, one page does not mix two GUI frameworks. When a page is migrated, we package it so that a packaged page includes all the code necessary to represent the page GUI. We detail how we propose to package pages in Section 3.2. It allows us to integrate the pages inside the architecture independently of their programming language. This tackles the *GUI mixing* challenge.

Yet, some data needs to be transferred between pages. For example, when navigating from one page to another, the source page might need to send data to the target one if a selected item in the source page

²<http://www.clifford.at/spl/>

³<https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe>

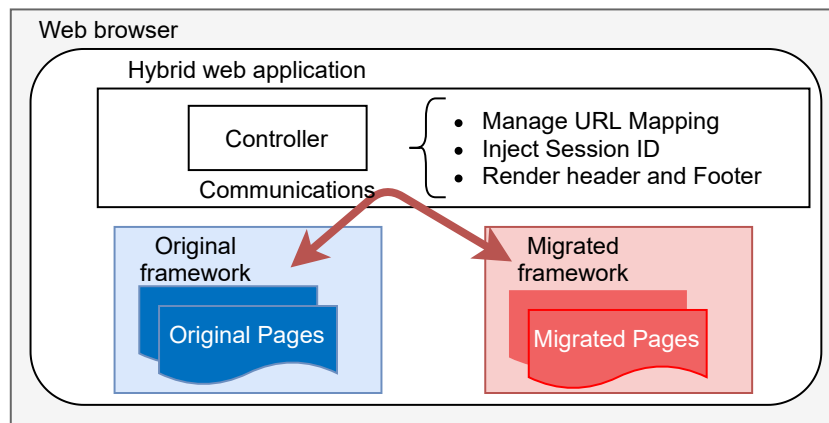


Figure 1: Hybrid architecture.

is detailed in the target page. Such a data transmission is done through the controller. The source page sends the data to the controller; then the other page pulls the data from the controller. The central position of the controller is thus used to tackle the *communication* challenge. Moreover, the controller checks whether the two pages use the same programming language. If not, the controller converts data before sending it. An implementation example of this process is described in Section 4.2. Using the controller as a mandatory gateway when transmitting data and enabling data conversion by the controller tackles the *type matching* challenge.

3.2 Packaging Pages using Web Components

Our hybrid architecture allows one to mix pages inside one web application. Since we are mixing pages in a web environment, all pages are rendered using the HTML, JavaScript, and CSS languages. Thus, to package pages for the hybrid architecture, we used Web Components⁴: reusable JavaScript modules including a markup structure (HTML), its associated script (JavaScript), and style (CSS). Any Web Component can be inserted into any web application independently of its original programming language and GUI framework.

In the following, we present how we package pages as Web Components and deploy them into the hybrid architecture. It consists of four steps.

1 – Select and Migrate Original Page. This step aims to select and migrate a page to the new framework. Ideally, the migration is (partly) automated (Verhaeghe et al., 2021), but this is not required. The

output of this step is the migrated page.

2 – Build the Web Component. This step takes as input a migrated page and builds a Web Component. Solutions for this already exist for many web GUI framework⁵.

3 – Register the Web Component. Every web browser has a registry that contains all the Web Components that can be used at runtime. We register each Web Component with an associated HTML tag. This tag will then be used to refer to the newly created Web Component.

4 – Replace Legacy Code by the Web Component Tag. The final step consists in modifying the original source code by replacing all the content of a page with the newly created Web Component tag. Thus, when end-users access the page, the browser injects the Web Component inside this tag in the HTML page.

Web Components allow one to insert pages inside an application defined with another GUI framework, answering the *GUI mixing* challenge.

We presented the main concepts to mix GUI of applications inside a hybrid architecture. Our hybrid architecture tackles the three major issues: *communication*, *type matching*, and *GUI mixing*. In the following, we detail an implementation for the migration of GWT applications to Angular and detail how it deals with the three issues.

4 IMPLEMENTATION

Our implementation allows one to mix GWT and Angular applications. We used it to enable the migration of GWT web pages to Angular ones. We first present the operational architecture (Section 4.1) dealing with

⁴Web Component: https://developer.mozilla.org/en-US/docs/Web/Web_Components

⁵For example: <http://www.gwtproject.org/doc/latest/polymer-tutorial/introduction.html>, or <https://angular.io/guide/elements#how-it-works>

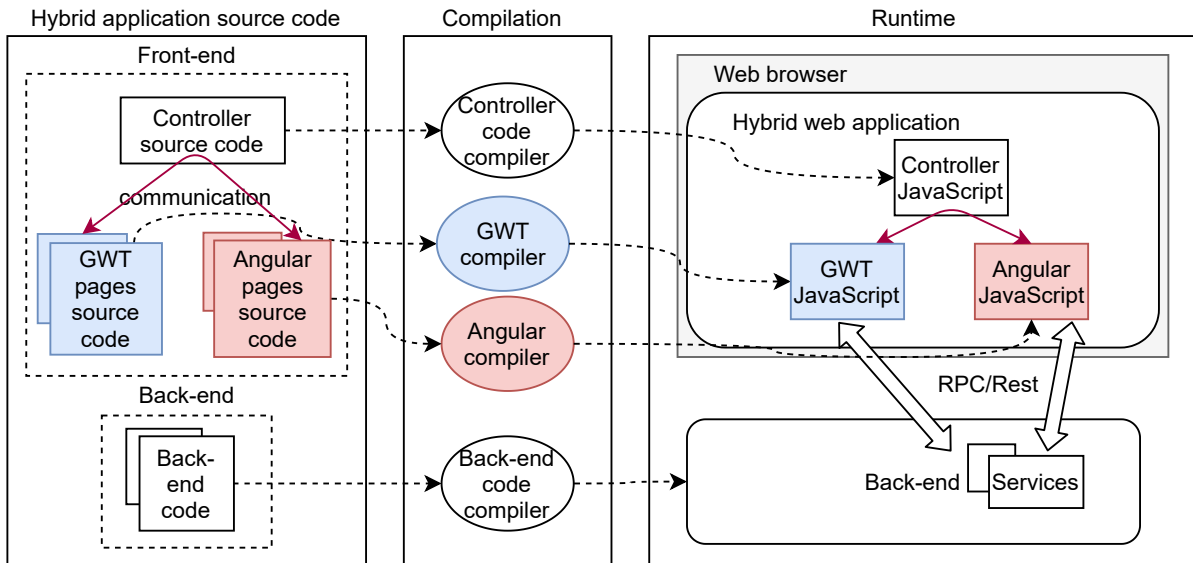


Figure 2: Operational architecture of hybrid application for GWT and Angular.

the *GUI mixing* challenge. Then we describe how we enabled communication in the architecture (Section 4.2) to deal with the *communication* and *type matching* issues.

4.1 Mixing GWT and Angular GUIs

We now present the operational architecture to mix GWT and Angular GUIs.

Figure 2 presents the operational architecture for a GWT and Angular hybrid application. The hybrid application source code (left) includes the front-end and the back-end parts. The front-end contains the source code of the GWT pages, the source code of the Angular migrated pages, and the source code of the controller. Note that the controller can be developed in any programming language. In our implementation, we used the existing GWT controller because switching to an Angular controller would require packaging every GWT original page as a Web Component and this is more tedious to do than for Angular. The back-end can be developed in any other programming language.

At compilation time (center), we use one compiler for each programming language. Thus, GWT pages are compiled using the GWT compiler, Angular with the Angular compiler, and the back-end with its own compiler (*i.e.*, in our case, Java).

At runtime (right), the compilers produce the hybrid application using our hybrid architecture presented Figure 1. Typically, web front-ends are compiled to JavaScript to be run in web browsers. This is the case for GWT and Angular. The hybrid web

application is also linked with the back-end.

When all pages are migrated, only the Angular pages are used. The back-end does not need to be migrated. The controller can be migrated in the target technology (*i.e.*, Angular in our example).

4.2 Enabling Communication between Angular and GWT Controller

In the hybrid architecture, all communications between pages go through the controller (see Section 3.1). For example, it is the case when one page displays information selected in another one. To do so, the pages need to communicate with the controller in charge of the data transmission. Since we use the pre-existing GWT controller, we need communication between the GWT pages and the GWT controller and between the Angular pages and the GWT controller.

GWT pages to GWT controller communication is trivially supported.

We faced two challenges to enable communication between Angular pages and GWT controller: *calling methods*, (instanciation of the *communication* challenge), and *sending/receiving data* (instanciation of the *type matching* challenge).

Calling Methods. To enable communication, we first need to allow Angular to call methods of the GWT controller. For example, Angular calls a method of the controller to navigate to another page or access some data. To enable method invocation, we studied the Angular and GWT compilers. We observed that they both translate source code (Java for GWT,

Typescript for Angular) into JavaScript code to be executed on the client-side (see Section 4.1). This is a common choice for web frameworks. Thus, both end up executed in the same programming language (i.e., JavaScript), in the same runtime (i.e., the web browser runtime). So, the Angular/JavaScript code has direct access to the methods of GWT/JavaScript. However, when GWT is compiled into JavaScript, the Java types and methods are normally not exposed externally, as names are mangled. Thus, they can not be used by other programs.

To expose Java methods and Java classes externally, GWT developers created the JSInterop library⁶. It allows developers to add annotations in their Java code that will tell the GWT compiler to expose the generated code to other programming languages by declaring the identifiers to use.

```

1 public class PhaseManager {
2     @JsMethod(name="displayPhase")
3     public void displayPhase(
4         PhaseMetadata pm) {
5         // ...
6     }
7     public void addData(String data) {
8         // ...
9     }
10 }

```

Listing 1: Exposed (displayPhase) and not exposed (addData) methods to hybrid architecture.

Listing 1 exemplifies this with one exposed method (displayPhase), through the use of the @JsMethod(<Name>) annotation (line 2), and one not-exposed (addData). Both methods are exported to JavaScript, but the not exposed one, addData, will have a mangled name making it impossible to be called from any hand-written code. On the contrary, the exposed method will get the name specified in the @JsMethod(<Name>) annotation. Using the JSInterop library, we ensure the *calling method* capability and tackle the *communication* challenge.

In the hybrid architecture for the GWT to Angular migration, we exposed 14 methods of the controller: nine for page navigation (including several options), two to check logged user rights, one to retrieve the currently opened page, and two to send and receive data during navigation and deal with the *type matching* challenge.

Sending/Receiving Data. We implemented a specific process to send and receive data between the

⁶<http://www.gwtproject.org/doc/latest/DevGuideCodingBasicsJsInterop.html>

pages and the controller. There are two cases: *classic communication* when data are sent and received by pages defined with the same GUI framework, i.e. GWT/GWT and Angular/Angular communication; and *hybrid communication* when data are sent and received by pages defined with different GUI frameworks, i.e. GWT/Angular and Angular/GWT communication.

For *classic communication*, a page (1) calls the navigation method of the controller with the data. Then, the controller (2) stores the data and opens the navigated page. The navigated page (3) asks the data to the controller, which, (4) returns it as it was stored.

For *hybrid communication*, the process is more complex due to the *type matching* challenge. Type matching is achieved through serialization and deserialization of the data. Because of its central position, this step is implemented in the controller.

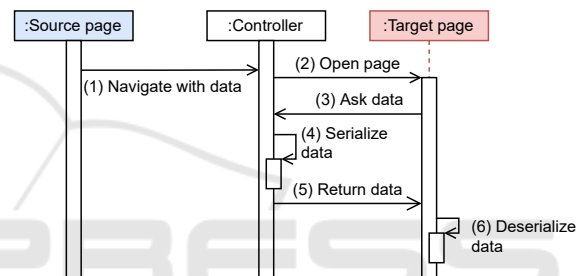


Figure 3: Data transformation process in hybrid communication.

To present the data transformation, we detail how the navigation between two pages is done. The data is sent by a source GWT page, goes through the controller, and is retrieved by a target Angular page (Figure 3). There are two exposed methods in the controller to retrieve the data: one for Angular, the other for GWT. Thus, based on the called method, the controller knows which framework, GWT or Angular, is retrieving the data. The controller also knows which framework sent the data by analyzing the data structure (Java instances coming from GWT, or plain JavaScript objects coming from Angular). Alternatively, two exposed methods to store the data could be created. Concretely, the source page (1) calls the navigation method of the controller with the data as a parameter. The controller stores the data as-is (i.e., does not perform any transformation) and (2) performs the navigation. Then, the navigated page (3) requests the data from the controller. Recognizing a hybrid communication, the controller (4) serializes the data in JSON and (5) returns it to the navigated page. Then, the navigated page (6) deserializes the data. Using this serialization/deserialization strategy, we dealt with the *type matching* challenge.

5 EVALUATION

5.1 Case Study: Omaje Application

To evaluate our hybrid architecture in a real industrial set-up, we used it to migrate a GWT application to Angular at Berger-Levrault. The application is called Omaje and was selected by its development team as representative of other Berger-Levrault GWT applications.

Omaje is a client subscription management application used internally and, therefore, a safe case study for our experiment. The Omaje application includes 20 main web pages distributed into 9 modules built using 6,683 GWT graphical elements. In total, in its original version, Omaje weights 191 KLOC in 2,669 Java classes and 14,882 methods.

5.2 Research Questions and Evaluation Methods

To evaluate our approach, we defined three research questions. For these RQs, we used three versions of the Omaje application: a GWT version, an Angular version, and a hybrid version with a controller written in GWT, three pages migrated to Angular and integrated into the application, and the remaining pages still in GWT.

RQ 1. Does data communication overhead impact the speed of browsing between pages?

This RQ aims to evaluate the impact of the serialization and deserialization approach we used to tackle the *type matching* problem. We expect that GWT to GWT and Angular to Angular communications are fast since they do not require additional data manipulation (*i.e.*, they are handled by the controller of the hybrid architecture but do not require serialization and deserialization). Hybrid communications might require additional time but need to be imperceptible to the end-user.

To evaluate this RQ, we execute scenarios requiring communications between GWT and Angular. The scenarios include the transmission of data of different sizes. Data includes objects with attributes of different types: primitives (*i.e.*, string, int, *etc.*), collections, dictionaries, and other objects.

We ran the scenarios in a simulated environment, *i.e.*, not performed by a real end-user, using Microsoft Edge version 87.0.664.66 on a laptop with 16 Go RAM and the Intel Core i7-7500U CPU. No other application was running on the computer during the experiment. To measure the communication time, we used the pre-built JavaScript feature `console.time()`.

To avoid bias in the computed times, each scenario is run 1,000 times, and we report the average time.

RQ 2. Does the build time of the hybrid and Angular applications deteriorate compared to the GWT one?

To evaluate this RQ, we measure the required time to build the GWT, the hybrid, and the Angular application. Building an application consists of creating the .class files and the transpilation of TypeScript (respectively Java) to JavaScript. Build time for large applications can be significant (hours), and it is important to ensure that building the hybrid application is not prohibitively long.

RQ 3. Does the GUI performance of the hybrid application deteriorate compared to the GWT original and Angular migrated applications?

This RQ aims to compare the performance when displaying the GUI of pages between the GWT, the hybrid, and the Angular applications. We measured the execution time of 4 execution scenarios for each application. The execution scenarios are:

1. Accessing the first page of the application where initial scripts are run;
2. Accessing a middle-sized page including about 100 widgets;
3. Accessing a large page that requests and displays a lot of data;
4. Modifying data with several requests to the database and updating the page UI.

When evaluating performance for the hybrid application, the accessed pages are in Angular, whereas the controller is in GWT. Thus, for the hybrid application, this research question evaluates if using Web Components tackles down the *GUI mixing* problem.

To measure the execution times, we used the built-in performance tool of Microsoft Edge browser⁷. It gives us detailed results on the GUI execution. We report the performance in scripting (executing JavaScript file), rendering (computing widgets position), and painting (displaying the resulting page) time.

5.3 Results

In the following, we present the result to the research questions.

RQ 1. Does data communication overhead impact the speed of browsing between pages?

We compute and report the average time of over 1,000 executions of communications between GWT and Angular in Table 2. During the communication,

⁷<https://docs.microsoft.com/en-us/microsoft-edge/devtools-guide-chromium/evaluate-performance/>

data of different sizes were exchanged, with several fields and cyclic references.

Table 2: Communications performance in millisecond (average on 1,000 executions).

	Target	
Source	GWT	Angular
GWT	2 ms	49 ms
Angular	7 ms	2 ms

For the communications between pages defined with the same GUI framework, both GWT-to-GWT and Angular-to-Angular communications need 2 ms. *Classic communication* through the controller does not require additional data manipulation.

For the *hybrid communications*, GWT-to-Angular communications require 49 ms, and Angular-to-GWT communications require 7 ms. So there is a cost for converting data. After investigation, we found that GWT-to-Angular communications are slower because the Angular deserialization library we used is less efficient than the GWT one. The poor performance of the Angular deserialization library is a known issue⁸ and might be solved in the future.

Summary: Hybrid communications cost does not impact the end-user. Although GWT-to-Angular communication is slower, it remains imperceptible for the end-user.

RQ 2. Does the build time of the hybrid and Angular applications deteriorate compared to the GWT one?

Table 3: Building performance in second.

Application	Building time
GWT	497 s
Hybrid	526 s
Angular	96 s

We measured the compilation time to build each application. Table 3 summarizes the time required to build the GWT, the hybrid, and the Angular applications. In GWT, there are two compilations: the build of the Java project and the Java to JavaScript transpilation when first accessing the application. Building the Java project costs 366 seconds, and the transpilation requires 131 seconds. Thus, the complete GWT compilation costs 497 seconds.

For the hybrid application, the build time is the same as for GWT (366 seconds). However, the transpilation time requires 160 seconds. The additional time for the transpilation comes from the usage of

⁸<https://github.com/pichillilorenzo/jackson-js/issues/18>

the JSInterop library that exposes the GWT controller methods. Thus, the hybrid architecture required 526 seconds.

The Angular application has a single compilation that requires 96 seconds.

Summary: This analysis shows that building the hybrid architecture is 6% slower than building the original GWT application. Thus, using a hybrid architecture has no important impact on building performance. Moreover, building the application is only done by developers, so end-users are not impacted by the building performance. It can be used to perform incremental migration.

RQ 3. Does the GUI performance of the hybrid application deteriorate compared to the GWT original and Angular migrated applications?

We compare the performance of each application for the four scenarios: home page first access, middle-size page first access, database request, and large page first access. Figure 4 presents the results of the performance evaluation. Each bar presents the time in milliseconds reported when evaluating the GWT application, the hybrid application, and the Angular application.

For the first access to the home page, the GWT application is the fastest. GWT pre-compiles the home page during compilation, thus, home page access is fast. On the other hand, when accessing the home page of the Angular application, the Angular runtime is loaded with the application script. This step is time-consuming for the Angular application. Finally, the hybrid application is the worst case because it combines the worst of both worlds. We investigated the reasons that make home page access slow when using Angular and discovered that our implementation does not use the *lazy loading* feature of Angular. Using this feature is part of our future work. Additionally, using the ahead-of-time⁹ compilation option of Angular could improve further the performances.

For the middle-size web page access, both the hybrid and the Angular applications are faster than the GWT one. The hybrid application benefits from the Angular speed, and the Web Components usage does not significantly deteriorate the time required to render the GUI. This scenario also highlights the low performance of GWT to render the GUI as compared to Angular.

The third group presents the performance of the application when users request the database (for instance, to update the data). Making a request implies a scripting step where the data are serialized, sent to

⁹<https://angular.io/guide/aot-compiler>

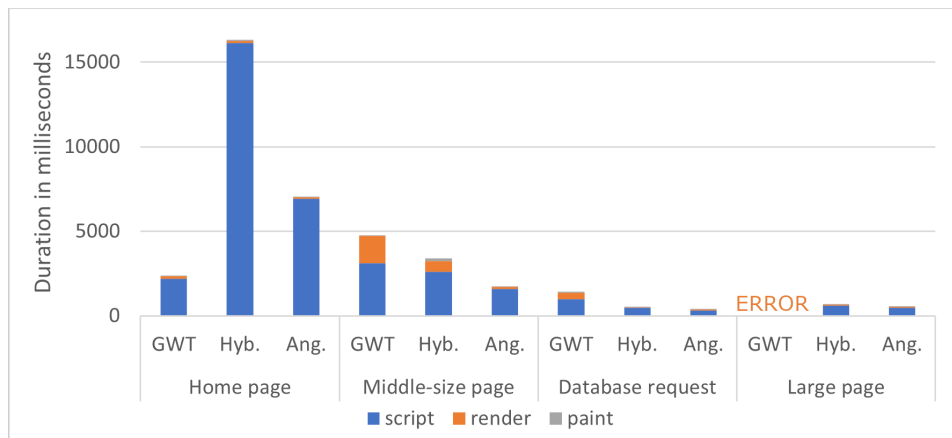


Figure 4: Performance evaluation result.

the back-end, new data are received, and the UI is updated. Again, the hybrid and the Angular applications are the fastest. It is due to a switch of communication protocol with the back-end and to the time required by GWT to *render* the new UI after retrieving the data.

Finally, the last group presents the performance when accessing a page displaying a lot of data. In the case of GWT, the Microsoft Edge built-in plugin crashed during the evaluation. So, we do not have execution time data. This is because GWT took several minutes to process the received data and the plugin ran out of memory. When performing the migration from GWT to Angular, the developers improved the performance of the page. In fact, they were able to use already optimized existing Angular widgets to fix this page. With the fix, the hybrid and Angular applications have better performances than the GWT one. We also note that thanks to the optimization, the large page became the fastest one.

Summary: Except for the first page access, the hybrid application has better performances than the original one. It is also the case of the migrated Angular application. The hybrid application takes benefits of the Angular features. RQ3 validates the usage of Web Components to tackle the *GUI mixing* problem.

6 THREATS TO VALIDITY

This section discusses the validity of our case study using the validation scheme defined by Runeson and Höst (2009). The construct validity, the internal validity, the external validity, and the reliability are presented.

6.1 Construct Validity

Construct validity indicates whether the studied measures really represent what is investigated according to the research questions. The purpose of this study is to evaluate the ability to use our hybrid architecture to migrate applications with GUIs.

For the developers, we validated that the build time of the hybrid architecture is not prohibitively long. We manually recorded the time needed to compile and transpile the application, which corresponds to the time spent by the developer waiting to see the application running. The scale of time required to build the application (more than 8 minutes), ensures that our conclusion is still valid, even considering some imprecision in the time manually recorded.

For the end-users, we evaluated the application usability against four scenarios. We reported the total time needed to perform the scenarios. To prevent possible bias, we selected four scenarios with different characteristics: size of the page, kind of request made to the database, and the first page in which initialization scripts are run. Time performances were recorded automatically and averaged over 1,000 runs to ensure the reliability of the results.

6.2 Internal Validity

Internal validity indicates whether no other variables except the studied one impacted the result.

Our validation is one industrial experiment consisting of the migration of a closed-source application. Even if we paid extra care to the tools we used to report the performance results of our hybrid architecture, it is rather difficult to isolate variables that might have impacted our results.

6.3 External Validity

External validity indicates whether it is possible to generalize the findings of the study.

We are aware that our results can not be easily generalized. We validated our incremental approach and its hybrid architecture against a closed-source application, and we can not publicly share the hybrid architecture implementation. Moreover, we describe a functional implementation to mix GWT and Angular GUI, but some issues might appear when mixing GUI defined with other GUI frameworks.

However, our hybrid architecture implementation is based on Web Components and the JSInterop library of GWT that are open-source projects. Thus, future research can easily reuse these projects for other hybrid architecture in a web context.

6.4 Reliability

Reliability indicates whether others can replicate our results.

Since our case study is a closed-source application, one can not replicate our result in the exact same context. Moreover, we do not provide any source code of the hybrid architecture.

To increase the reliability of our results, we perform the validation using standard free-to-use tools such as Microsoft Edge and pre-built JavaScript features. We also detailed our evaluation methods to ease future researchers reproducing the same evaluation set-up.

7 CONCLUSION AND FUTURE WORK

In this paper, we proposed a hybrid architecture that enables the migration of large and complex GUI applications. In particular, it deals with the *communication*, the *type matching*, and the *GUI mixing* issues.

The hybrid architecture can be reused for other web-based GUI migrations. Web Components can be used for other migration projects since it is nowadays a standard feature. However, future research should pay attention to the *communication* and the *type matching* issues that still might require specific work depending on the GUI frameworks.

We implemented our hybrid architecture to enable the integration of Angular pages inside a GWT application. It allows us to successfully perform the migration of an industrial project and gives good performance results.

Future work includes the implementation of our hybrid architecture to perform other GUI migrations. For example, we plan to migrate a large AngularJS application to Angular. We also plan to enable Angular features such as lazy loading and ahead-of-time compilation and evaluate their impact on performance.

REFERENCES

- Chisnall, D. (2013). The challenge of cross-language interoperability. *Communications of the ACM*, 56(12):50–56.
- Comella-Dorda, S., Wallnau, K., Seacord, R. C., and Robert, J. (2000). A survey of legacy system modernization approaches. Technical report, Carnegie-Mellon univ pittsburgh pa Software engineering inst.
- Flores-Ruiz, S., Perez-Castillo, R., Domann, C., and Puica, S. (2018). Mainframe migration based on screen scraping. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 675–684. IEEE.
- Kontogiannis, K., Martin, J., Wong, K., Gregory, R., Müller, H., and Mylopoulos, J. (2010). Code migration through transformations: An experience report. In *CASCON First Decade High Impact Papers*, pages 201–213. Unknown.
- Polito, G., Ducasse, S., Tesone, P., and Brunzie, T. (2020). Unified ffi - calling foreign functions from pharo.
- Robillard, M. P. and Kutschera, K. (2019). Lessons learned while migrating from swing to javafx. *IEEE Software*, 37(3):78–85.
- Runeson, P. and Höst, M. (2009). Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical software engineering*, 14(2):131–164.
- Sánchez Ramón, O., Sánchez Cuadrado, J., and García Molina, J. (2014). Model-driven reverse engineering of legacy graphical user interfaces. *Automated Software Engineering*, 21(2):147–186.
- Sneed, H. M. et al. (2006). Wrapping legacy software for reuse in a soa. In *Multikonferenz Wirtschaftsinformatik*, volume 2, pages 345–360. Citeseer.
- Tepe, W. (2009). The arno project: Challenges and experiences in a large-scale industrial software migration project. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 149–158. IEEE.
- Terekhov, A. A. and Verhoef, C. (2000). The realities of language conversions. *IEEE Software*, 17(6):111–124.
- Verhaeghe, B., Anquetil, N., Etien, A., Ducasse, S., Seriai, A., and Derras, M. (2021). GUI visual aspect migration: a framework agnostic solution. *Automated Software Engineering*, 28(2):6.
- Zhang, B., Bao, L., Zhou, R., Hu, S., and Chen, P. (2008). A black-box strategy to migrate gui-based legacy systems to web services. In *2008 IEEE International Symposium on Service-Oriented System Engineering*, pages 25–31. IEEE.