Quarmic: A Data-Driven Web Development Framework

Pedro Miguel Pereira Cunha
CRACS & INESC Tec LA / Faculty of Sciences, University of Porto, Portugal
up201405950@fc.up.pt

José Paulo Leal
CRACS & INESC Tec LA / Faculty of Sciences, University of Porto, Portugal
`http://www.dcc.fc.up.pt/~zp`
zp@dcc.fc.up.pt

Abstract

Quarmic is a web framework for rapid prototyping of web applications. Its main goal is to facilitate the development of web applications by providing a high level of abstraction that hides Web communication complexities. This framework allows developers to build scalable applications capable of handling data communication in different models, data persistence and authentication, requiring them just to use simple annotations. Quarmic's approach consists of the replication of the shared object among clients and server in order to communicate through its methods execution. Where the annotations, namely decorators, are used to indicate the concern (model or view) that each method addresses and to implement the framework's inversion of control. By indicating the method concern, it enables the separation of its execution across the clients (responsible for the view) and the server (responsible for the model) which facilitates the state management and code maintenance.

2012 ACM Subject Classification Software and its engineering → Development frameworks and environments; Software and its engineering → Application specific development environments

Keywords and phrases web development, framework, data-driven

Digital Object Identifier 10.4230/OASIcs.SLATE.2019.19

Funding This work is partially funded by the ERDF through the COMPETE 2020 Programme within project POCI-01-0145-FEDER-006961, and by National Funds through the FCT as part of project UID/EEA/50014/2013.

1 Introduction

The emergence of Web 2.0 and the popularization of mobile apps with internet connectivity broaden the scope of web applications thus increasing the demand of interactive applications that support a high number of users and deliver a real-time experience. As a result, web development has become more difficult, because the range of problems that developers have to deal have also increased, such as concurrency and scalability, or security and session management issues.

This paper presents Quarmic, a web development framework for rapid prototyping of web applications built on top of Node.js. Specifically, it consists of a high-level tool that facilitates deployment of web apps with a sophisticated communication system, capable of handling data communication in different models (broadcast, multicast and unicast), data persistence and authentication in a transparent and scalable way. The main goal of Quarmic is to simplify the development of web applications by raising the level of abstraction. A high level of abstraction enables developers, with little expertise on web communication, to build complex web apps, since it completely frees the developers from implementing any component regarding communication. Consequently, it also allows them to focus on other parts of the application, such as the user interface and the logic itself. Web applications built with

Quarmic follow a class-based object-oriented paradigm in which each class and its methods can be annotated using the annotation syntax provided by the framework. This annotation syntax is the mechanism used by the developer to assign a particular role to each method (a model or view concern) since the communication is fully based on methods execution. Also, the objects of the annotated classes are shared among the server and clients, allowing to keep the server's object as the single-source-of-truth. Which implies the separation of the method execution by its concern between the server and the clients. Hence, once the objects are instantiated, Quarmic inverts control thus implementing and controlling the execution of its methods (including persistence and authentication).

As a result, developers can create web apps as shared objects. For instance, Quarmic can be used to prototype a real-time multiroom chat. Where the chat itself is a shared object that contains a list of rooms, which are also shared objects. Every single client will share the chat object, but only the room participants will share the object that represents its room. Another example can be a collaborative spreadsheet app, where each cell is a shared object, allowing to maintain the state of each cell or a group of cells separately.

This novel approach is an advantage relative to the existing frameworks since it only requires a few annotations to implement in contrast with the coding required by any other approach. Therefore, because of its level of abstraction, a developer acquainted with the oriented object paradigm will be able to quickly gain web development proficiency. In addition, Quarmic does not restrict the use of other framework or libraries to program other tiers of the application, which is advantageous to the flexibility of programming.

The remainder of this paper is organized as follows. Section 2 reviews the architecture of Node.js and some features of a few web application frameworks that influenced the creation of this framework. Section 3 describes Quarmic's architecture as well as its main features, presenting some implementation details. Section 4 concludes the purpose of this paper and summarizes the future work.

2 State-of-the-Art

Node.js & Javascript

In the past years, Node.js has becoming increasingly popular among developers [4], as a result of its architecture and its modules that allow to quickly and effectively build highly scalable Web applications. Node's architecture introduces event-driven programming to the server-side scripting, which is a much more efficient approach in terms of memory when dealing with concurrency. Despite not being the first platform to do this, it is by far the most successful [5]. Its asynchronous event-driven JavaScript (single-threaded) runtime provides a preferable application environment to develop highly scalable systems [2].

Another contributing factor to the Node's adoption as the preferred platform to application development is the single language feature. Node enables client-side and server-side scripting in Javascript, elevating the language, which formerly was just used as a client-side language, to a new height of server-side web development [1]. Therefore, it simplifies the development environment, since developers can build the entire tier of their applications in the same language. An approach that Google Web Toolkit framework introduced, where developers write all components in Java. Then, the browser components would be compiled to Javascript and HTML.

Web Application Frameworks

Web application frameworks are software frameworks that make it easier to build web applications. They provide tools and libraries that simplify common web development tasks, such as HTML generation, state and session management and database interaction.

Frameworks such as Ruby on Rails[1] or Laravel[2], provide an Object-Relational Mapping (ORM) layer that simplifies the use of relational databases in Web applications by mapping each database table with a class in the underlying language thus relieving developers of the hassles of dealing with the underlying database [8]. Another common feature is its architecture. Both follow the Model-View-Controller (MVC) pattern that divides the applications into three interconnected parts. The Model, that manages the business logic; the View, that defines the presentation of the Model to the user interface; and the Controller, that serves as an intermediary between the View and Model, responding to the user input and interactions [7].

Other frameworks, such as Vue.js[3], offer state management features. Vues' reactivity system makes state management simple and intuitive. When a JavaScript object is passed to a Vue instance, all properties are converted to getters / setters using `Object.defineProperty`. These getters/setters are hidden by an abstraction layer that makes them invisible to users, but internally they allow you to run dependency-tracking and change-notification whenever the properties in question are accessed or changed. Each instance of a Vue component has a corresponding watcher instance. During component rendering, this watcher instance registers, the properties converted to getter as dependencies. As such, if a setter of a dependency is executed, it will notify the watcher, which will cause the re-rendering of its component [9].

3 Quarmic

Quarmic is a Node.js framework for *rapid prototyping* of web applications. Its main feature is the support for data communication between multiple clients. As a web framework, its goal is to facilitate the web application development process. This framework is aimed at web applications that deal with near real-time behaviour, in particular, applications that share objects among multiple clients. It accomplishes this goal by taking control over the execution flow of the application, which provides the ability to sequence and coordinate the application activity in order to receive and process data in a consistent way, and return results quickly enough to ensure a near real-time behaviour. Thus, it frees the programmer from the implementation of these tasks, allowing them to focus on the application custom logic and appearance.

The main differentiation feature of Quarmic is its architecture 1. It follows the object-oriented paradigm to design the application, allowing the developers to create *shared objects*, objects that are shared among different clients. Unlike the majority of the web application framework, this framework doesn't rely on the Model-View-Controller (MVC) pattern to assign different web app components to each of those roles. Instead, it enables the association of those roles to methods of the corresponding classes by annotating them according to the concern they address, which can be either a business logic operation (model) or a presentation logic operation (view).

The approach to share an object relies on the replication of the server's instance. Once the server instantiates a shared object, this object becomes shareable among all clients. This means that clients that instantiate it will share it with the server and the other clients that have instantiated it as well. And since the client's instance is created as a replica of the

[1] `https://rubyonrails.org/`
[2] `https://laravel.com/`
[3] `https://vuejs.org/`

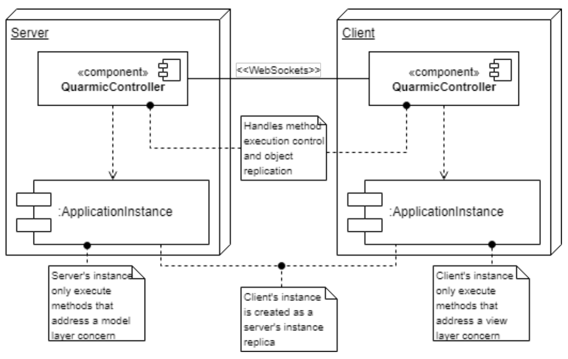
Server
«component»
QuarmicController
<<WebSockets>>
Client
«component»
QuarmicController
Handles method execution control and object replication
:ApplicationInstance
:ApplicationInstance
Server's instance only execute methods that address a model layer concern
Client's instance is created as a server's instance replica
Client's instance only execute methods that address a view layer concern

Figure 1 Quarmic's architecture.

server's object, the framework does not allow clients to instantiate a shared object without it had been previously instantiated in server. By performing a distributed execution, the framework is able to control the method execution of these objects, ensuring that methods that have been annotated as addressing a model concern are only executed by the server's instance and methods that have been annotated as addressing a view concern are only executed by the clients' instance (server's instance replica). Moreover, the framework updates the clients' replica before executing any method.

Therefore, by annotating methods with the concerns they address, classes declarations can be shared among server and clients, facilitating data management. Since the server is the only component allowed to change the object state, an object that is shared among multiple clients maintains a consistent state. The fact that this framework is built on top of Node.js provides a JavaScript runtime environment that allows the application to be coded in a single language, JavaScript. This approach is also compatible both with popular client-side cross-browser JavaScript libraries and toolkits, such as Bootstrap or JQuery and with Node.js modules.

In the following subsections, the framework is described in more detail. Firstly, the annotation syntax is presented. Secondly, the inversion of control of the framework is addressed by presenting its core processes and some implementation details.

3.1 Annotation Syntax

The following code snippet presents a class with Quarmic annotations. In this example, the class `Counter` encodes a counter with a button to increment it. The counter's initial state is set by the class annotation and its two methods are annotated according to their role. The `increase` method increments the counter value (object state modification) and calls the `update` method, thus addressing a business logic concern. Conjointly, the `update` method addresses a view concern, since it updates the counter value in the user interface. Incidentally, in this example, the controller role is provided by the constructor since it binds user actions to the model, but this is not relevant to Quarmic.

```
@sharedProperties({value: 0})
class Counter{
constructor(){
this.counterElem = document.getElementById("counter-value");
this.incElem = document.getElementById("increase-button");
this.incElem.onclick = this.increase.bind(this);
this.update.call();
}
@cause
increase(){
this.value++;
this.update();
}

@effect
update(){
this.counterElem.value = this.value;
}
}
```

Quarmic's annotations use the ES7 decorator proposal, which enables the modification of JavaScript classes and properties at compile time without explicitly modifying them [3]. Decorators allow programmers to apply the desired behaviour to the application built with Quarmic. As mentioned before, they inform the framework of the concern addressed by the methods. But more importantly, they implement Quarmic's inversion of control, which gives the ability to take control over the application lifecycle in order to perform the object replication among clients and server and to control both method execution.

The code is influenced by annotations according to the side where it is executed. Thus, a decorated class on the server-side will behave differently from the same decorated class on the client-side. The main Quarmic annotations are the following.

`@sharedProperties(properties)`

This annotation is applied to a class declaration to specify the properties (fields) of shared objects and their initial values. Essentially, it modifies the constructor in order to be used as a dependency injection container. On the server-side, the constructor assigns the shared object state properties passed by the 'properties' variable (list of properties) of the argument and sets up the database facility. On the client-side, the constructor performs the object replication (process described in the following section) before executing the original code.

`@cause(auth)`

Indicates to the framework that the method annotated with this decorator addresses a model concern, delegating its execution if the method is invoked on the client. Also, if the "auth" variable is passed to the decorator as "required", the cause will require authorization in order to be executed. Otherwise, if it is passed as "none" or it is omitted, the cause method will no longer require authorization to be executed. This argument is only used when the class is protected by authentication (process described in the following section).

`@effect(scope)`

Indicates to the framework that the method annotated with this decorator addresses a view concern, delegating its execution if the method is invoked on the server. Also, if the "scope" variable is passed to the decorator as "private", the effect will only be executed in the client that has invoked the cause method of that effect. Otherwise, if it is passed as "public" or it is omitted, the effect method will be executed by all clients.

3.2 Distributed Execution

Quarmic's architecture is based on a distributed execution. This process relies on WebSockets to keep a continuous exchange of data between the clients and the server in order to perform the object replication and to control the execution of methods. It is implemented using the Socket.io library, which enables near real-time, bidirectional and event-based communication between server and clients.

The following subsections address how the methods execution control and the object replication are performed.

3.2.1 Object Replication

Object replication consists in pushing the state of the server's object to their client's replicas. This process occurs when the object is instantiated and establishes a socket to the server, which is used to request the up-to-date state. As a result, the client's object will inherit the server's object properties, transforming the client's object into a replica of the server's object. The following figure 2 illustrates this process.

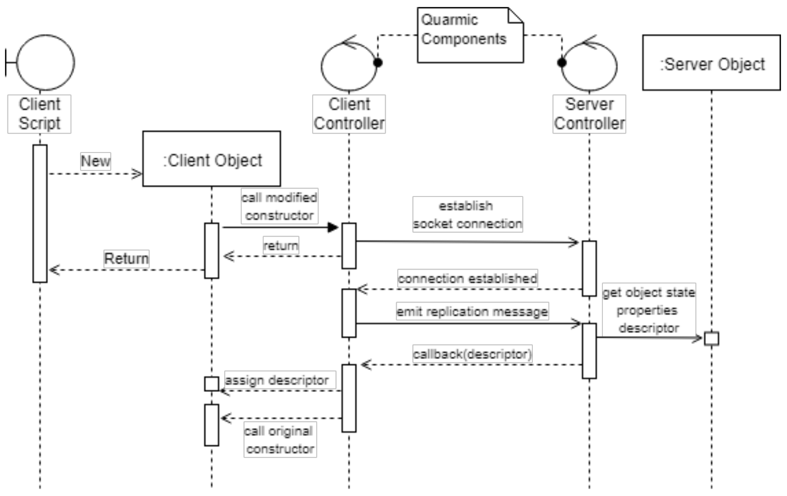

Figure 2 Sequence Diagram - Object Replication.

3.2.2 Method Execution Control

As previously referred, the method execution control is intended to ensure that the model layer methods are only executed in the server and the view layer methods are only executed in the client. It is inspired in the *causality principle* that states that for every *effect* there is a definite *cause*. Hence, a view method execution (effect) is always preceded by a model method execution (cause). Therefore, the framework will not allow clients to directly invoke an effect method. Both server and clients can directly invoke cause methods. For instance, we can have an application with a web service running in the server that calls cause methods, and components in the user interface bound to cause methods, but only the server can execute them. Moreover, it's important to note that this relationship can be chained. In particular, a cause method can be called within another cause method or within an effect method.

This approach allows to maintain a consistent state among multiple clients. As the server executes a cause method, invoked by itself or by one of the clients, it propagates the up-to-date state and the effects methods execution instructions to each one of the clients.

Eventually, each one of them will update its object's state and execute the effects, updating the object visual representation in an accurate and consistent way. . The following 3 figure illustrates this process.

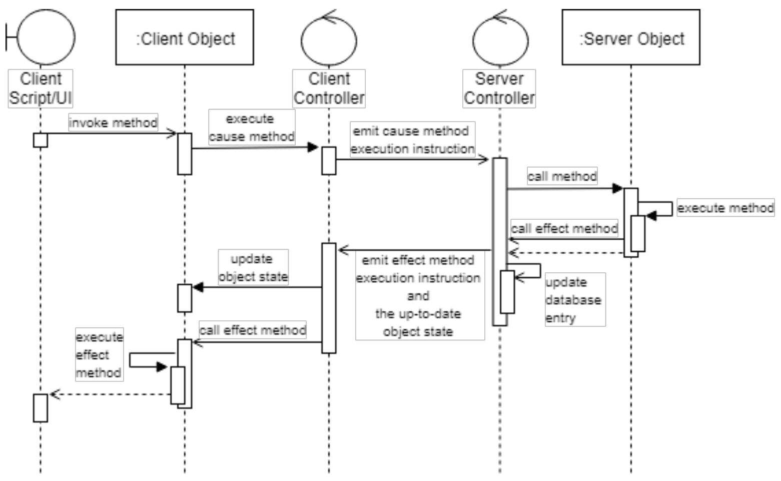

Figure 3 Sequence Diagram - Method Execution Control.

Since JavaScript has a concurrency model based on an event loop, it ensures that whenever a method is executed, it cannot be pre-empted[6]. This enables the server to handle multiple concurrent calls of cause methods by multiple clients because the server is able to coordinate the execution of those methods even if they were called almost simultaneously. And the same goes to the propagation of the object state and the resulting effects.

3.2.2.1 Data persistence

Quarmic handles data persistence by itself. When the server instantiates a shared object, the framework creates an entry in the database to store the object state or to get the object data if already exists. This entry is updated whenever the object state is modified, ensuring an up-to-date backup of the shared object. Since this process is done asynchronous, it won't overload the server. Moreover, to improve performance, up to a certain number of objects are kept in memory, using a least recently used (LRU) cache.

3.2.2.2 Authentication

Quarmic also provides an authentication feature. It consists of a token-based authentication system and is also controlled through the invocation and execution of methods. These methods are a special type of cause (annotated with the `@authenthicator` decorator) and they are responsible for assigning a token that initiates the client's authorized session in the shared object domain.

When a class has a method of this kind (authenticator method), it will be treated as a protected class, which means that all others methods will require authorization to be invoked. The authorization is provided by the authenticator method and since it addresses a model concern, it is executed in the server. If its execution determines that was successfully authenticated, the framework assigns a token to the clients' object, establishing an authorized session. This is useful in methods that implement login, for instance.

4 Conclusions and future work

This paper presents Quarmic, a web application framework that aims to facilitate the development of applications, in particular, near real-time web applications. The framework includes a high-level mechanism that supports real-time applications as a single shared object (e.g. a webchat) or a set of shared objects (e.g. a collaborative spreadsheet app).

Quarmic is a work in progress, currently in the final development stage. At the time of writing, it lacks some improvements in the deployment of the applications and validation regarding system testing and user acceptance testing. The main challenge it has been the implementation of the facility to deployment because at this moment the decorators' proposal is at stage 2 (Draft), which is a hindrance to work with them in the browser. The current workaround is the transpilation of the decorators to ES5. The remaining steps go through to test its performance in the real world by performing some stress tests and user acceptance tests.

References

1 Dave Anderson. How Node.js can accelarate development, 2014. Modulus.

2 Nimesh Chhetri. A Comparative Analysis of Node.js (Server-Side JavaScript), 2016. Culminating Projects in Computer Science and Information Technology. Paper 5.

3 JavaScript Decorators. [Online; accessed April 2019]. URL: `https://github.com/tc39/proposal-decorators`.

4 Node.js Foundation. Node.js User Survey, 2018. [Online; accessed April 2019]. URL: `https://nodejs.org/en/user-survey-report`.

5 Tom Hughes-Croucher and Mike Wilson. *Node: Up and Running*. O'Reilly Media, Inc., 2012.

6 Neelakantan R. Krishnaswami Jennifer Paykin and Steve Zdancewic. The Essence of Event-Driven Programming, 2016. Unpublished Draft. URL: `https://www.cl.cam.ac.uk/~nk480/essence-of-events.pdf`.

7 Abdul Majeed and Ibtisam Rauf. MVC Architecture: A Detailed Insight to the Modern Web Applications Development, 2018.

8 David B. Copeland Sam Ruby and Dave Thomas. *Agile Web Development with Rails 5.1*. Pragmatic Bookshelf, 2017.

9 Reactivity in Depth. [Online; accessed April 2019]. URL: `https://vuejs.org/v2/guide/reactivity.html`.