An Architecture Pattern for Network Resilient Web Applications

Julien Richard-Foy, Olivier Barais, and Jean-Marc Jézéquel
{first}.{last}@irisa.fr

IRISA, Université de Rennes, France

Abstract. With the advent of Internet connected mobile devices having a an intermittent network connection rises the need for a good support of network resilience for Web applications. Achieving resilience requires to resite the application logic execution on client-side when network failures are detected and then to synchronize changes to the server when the connection is back. These failures can happen on each possible user action, making resilience a cross-cutting concern, and therefore making it hard to isolate in the application's code. This paper presents an architecture pattern based on event sourcing isolating the network resilience concern. Applications built on top of this architecture pattern keep delivering a good quality of experience under bad network conditions. We implemented two example applications and captured the client-server synchronization logic as a reusable library.

1 Introduction

Web applications work only as long as the network connection and the server are up. If the network connection or the server is down, applications may suffer from inconsistency, freeze or latency, resulting in a bad user experience. With the advent of Internet connected mobile devices (smartphones and tablets) rises the need for a good support of an off-line mode for Web applications.

In this paper, we use the term *resilience* to refer to the ability of a Web application to deliver a good Quality of Experience (QoE) under bad network conditions. The resilience level can vary. For instance, when an application detects a network failure, it can either:

- Display an error message informing the user that its last action failed (*e.g.* Trello[1]);
- Fallback to a read-only mode (*e.g.* Google Drive[2]);
- Let the user work and try to synchronize the changes when the connection is back (*e.g.* Workflowy[3]).

[1] http://www.trello.com
[2] http://drive.google.com
[3] http://www.workflowy.com

To keep delivering a good QoE when the server is unreachable, the application must resite the business logic on the client-side and then synchronize the changes to the server when the connection comes back. According to the application's domain this task can be more or less hard to achieve. Typically, if the logic needs to access to a large amount of data or to perform a heavy computation, the price to pay may be too high for the client. However, in all other cases the client could take on the business logic execution. Nevertheless, the need for detecting and handling network failures on all user actions makes resilience a cross-cutting concern [6], and makes it hard to isolate in the application's code.

This article presents an architecture pattern based on event sourcing to isolate the resilience concern in Web applications. Our pattern is agnostic to the whole application architecture, it requires little adaptation for existing applications to use it, it only captures and isolates the client-server execution distribution policy and allows applications to deliver a good QoE when the network is down.

Based on this architecture pattern we implemented libraries for both the server and client sides, and used it to develop two use case applications demonstrating the isolation of the resilience concern.

2 Related Work

[9]

2.1 Offline Support

[8], [7], [5].

2.2 Persistent Resources

Cannon *et. al.* showed how to get Web application working offline using persistent resources [3]. This approach is *data centric*: data objects are enhanced with a persistence and synchronization mechanism so that automatically they keep track of their changes and synchronize themselves to a server.

[1]

2.3 Event sourcing

Event sourcing "captures all changes to an application state as a sequence of events" [4]. Event sourcing is mostly used in conjonction with command and query segregation in order to solve scalability problems [2].

3 Contribution

Our architecture relies on the idea of event sourcing: the state of the application is determined by the application of a succession of events. For instance, in the case of a text editor, such events could be text insertion and text deletion. Events are just described by data: *e.g.* the position and the character inserted.

Essentially, the idea behind our architecture is to store the events corresponding to each performed action in a queue, on client-side, and to synchronize the queue content when the network connection is up. The business logic corresponding to each event is executed on client-side so users do not suffer from latency when they interact with the application.

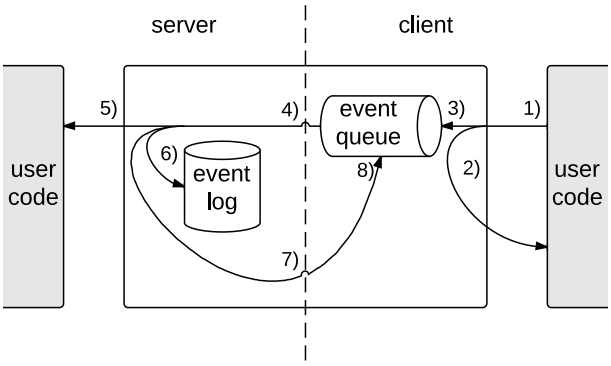

Fig. 1. Architecture

Figure 1 depicts the process involved when an user triggers an action in the application. The application code is partitioned between the client and server side, with a synchronization module in between. The left and right parts, in gray, represent the application code, and the central part represents the synchronization module. The process involved when a user triggers an action is the following:

1. A domain event corresponding to the user action is created on the client and sent to the synchronization module ;
2. The synchronization module calls back the logic defined by the application for this event ;
3. The event is put into an event queue ;
4. If the network connection is up, the queue content is sent to the server side part of the synchronization process ;
5. The synchronization module invokes the business logic defined by the application for this event ;
6. The event is stored in an event log on the server side ;
7. The application of the event is acknowledged to the client ;

8. The client removes the acknowledged event from its queue.

With this design, users have an instant feedback for their actions because their logic is executed on client-side.

Figures 2 and 3 show a class diagram of our architecture pattern. Classes in gray are those written by users, classes in white are provided by our library. The application defines the `Event1` and `Event2` domain events and implements the business logic corresponding to their interpretation on client-side and server-side (methods `interprete` of classes `server.AppSync` and `client.AppSync`, respectively). When an user performs an action the `client.Sync#apply` method is called, which calls back the logic implemented in `client.AppSync.interprete`, put the event in its queue and tries to synchronize the queue to the server. On server-side, the `server.Sync.join()` method is called when a client joins the application, it returns both a sink and a source of events. The sink is used when events are received from a client and the source is used to notify a client that an event has been logged. When an event is pushed to the sink, the `interprete` method is called, applying the business logic corresponding to the event, the event is appended to a log (class `Log`), and pushed back to the client *via* the source. The classes drawn with dotted lines are not part of the architecture pattern, they are given here as an illustration of a possible integration with the whole application. Here, we have a class `State` on server-side, keeping an in-memory state of the system, and classes `Model`, `Ctl` and `View` on client-side, following a kind of model-view-controller pattern.

4 Validation

4.1 TodoMVC application

4.2 Notes application

4.3 Limitations

5 Discussion

References

1. Edward Benson, Adam Marcus, David Karger, and Samuel Madden. Sync kit: a persistent client-side database caching toolkit for data intensive websites. In *Proceedings of the 19th international conference on World wide web*, WWW '10, page 121130, New York, NY, USA, 2010. ACM.
2. Dominic Betts, Julian Dominguez, Grigori Melnik, Fernando Simonazzi, and Mani Subramanian. *Exploring CQRS and Event Sourcing: A Journey into High Scalability, Availability, and Maintainability with Windows Azure.* Microsoft patterns & practices, 1st edition, 2013.
3. Brett Cannon and Eric Wohlstadter. Automated object persistence for javascript. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 191–200, New York, NY, USA, 2010. ACM.

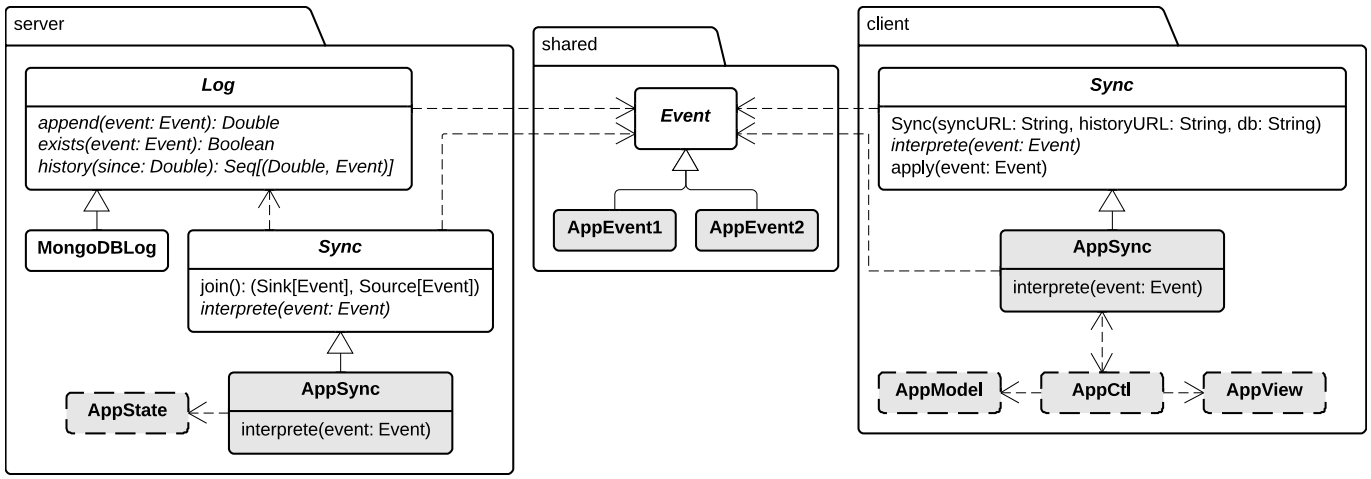

Fig. 2. Architecture

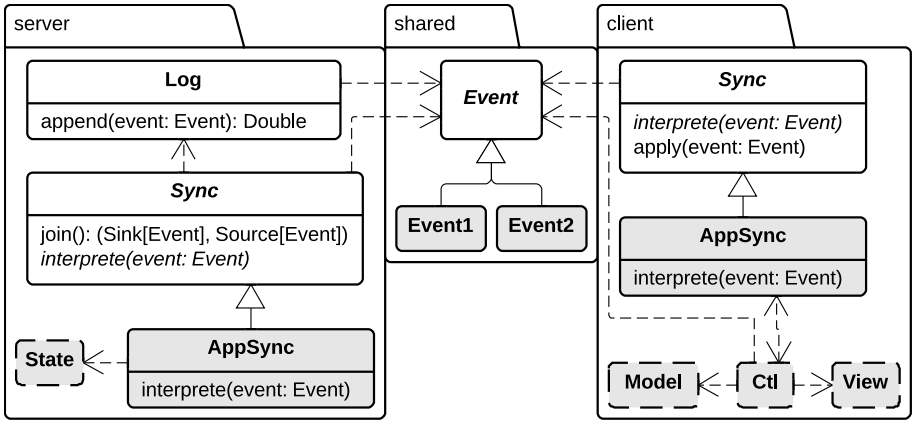

Fig. 3. Architecture

4. Martin Fowler. Event sourcing. *Online, Dec*, 2005.
5. Yung-Wei Kao, ChiaFeng Lin, Kuei-An Yang, and Shyan-Ming Yuan. A Web-based, Offline-able, and Personalized Runtime Environment for executing applications on mobile devices. *Computer Standards & Interfaces*, 34(1):212224, 2012.
6. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akit and Satoshi Matsuoka, editors, *ECOOP'97 Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin Heidelberg, 1997.
7. Flix Albertos Marco, Jos Gallud, Victor Penichet, and Marco Winckler. A Model-Based Approach for Supporting Offline Interaction with Web Sites Resilient to Interruptions. 2011.
8. Yun Yang. Supporting online Web-based teamwork in offline mobile mode too. In *Web Information Systems Engineering, 2000. Proceedings of the First International Conference on*, volume 1, page 486490. IEEE, 2000.
9. Xinwen Zhang, Sangoh Jeong, Anugeetha Kunjithapatham, and Simon Gibbs. Towards an elastic application model for augmenting computing capabilities of mobile platforms. In *Mobile wireless middleware, operating systems, and applications*, page 161174. Springer, 2010.