Synchronized Symmetric Model-View-Controller

Michele Sama[1] and Franco Raimondi[2]

[1] PuzzleDev s.n.c.
Via A. Badiali 140, Ravenna, Italy
m.sama@puzzledev.com
[2] School of Science and Technology
Middlesex University, London, UK
f.raimondi@mdx.ac.uk

Abstract. In the past two decades the Model-View-Controller pattern has been employed successfully in the development of software systems. In this paper we argue that this model may be improved to support the development of applications running on multiple devices, possibly not always connected. Specifically, we introduce the notion of Symmetric Synchronized Model-View-Controller, in which multiple views and controllers are generated from a single model, and we propose the adoption of synchronization mechanisms based on ideas borrowed from the approaches developed for collaborative software (groupware) communication.

1 Introduction

The Model-View-Controller (MVC) pattern (see for instance the standard reference [1] for a detailed description) has been employed successfully in the past two decades for the development of software systems. A number of development frameworks and development environments currently in use support and encourage the adoption of this pattern. Notable examples include Apple XCode for the development of iOS applications and the open source frameworks Django [2] and Spring [3] for the development of Python and Java applications, respectively. Typically, in these frameworks developers specify a model and the code for the views and the controllers can be partially generated in an automatic way. For instance, a web-based application can be generated in Django by describing the model (essentially, an abstraction of the data model), and the corresponding views and controllers are automatically generated for clients using a browser[3].

Due to the increase in the number of mobile platforms, web based applications are required to be able to interact with a variety of clients, in addition to "standard", desktop-based clients. In many circumstances it may be required that mobile clients are also able to work off-line. A typical example are web-based calendars, which can be accessed by a (connected) desktop, and by a a range of

[3] https://docs.djangoproject.com/en/dev/topics/class-based-views/generic-display/

native mobile applications that make a local copy of the remote database and present the former when connectivity is not available.

The current trend in industry is to consider the server-side (together with its browser-based client) and the mobile platforms as different applications all together. In fact, in some cases these products are developed by different companies, specialising in different technologies, e.g. Django for the server-side[4], and the iOS SDK for the mobile clients. This means that developers need to work on multiple MVC patterns, one for each platform, which need to be kept aligned when a change is made in the server model, and with the additional issue of synchronization of the various local databases. As exemplified in the following sections, this causes substantial code repetition and makes validation and verification impractical.

In this paper we argue that server and all the mobile clients should be seen as a *single modular application*, in which the application for each platforms *depends* on the applications running in all the other platforms, and this dependency is exploited at the modelling stage to automatically generate code. Our idea is to extend the MVC pattern by introducing the notion of Symmetric Synchronized MVC (SSMVC):

1. Symmetric: extract views and controllers for multiple classes of devices from a single model (the server model).

2. Synchronized: automatically generate code for client-server synchronization using ideas borrowed from Computer Supported Cooperative Work (CSCW, e.g., DiscoTech [4])

We provide the details of this idea in the rest of the paper, which is organised as follows: in Section 2 we describe a scenario that is used throughout the paper as a running (and motivational) example, while our idea is described in Section 3. We provide related work in Section 4 and we conclude in Section 5.

2 A motivational example

In this section we describe a scenario that is obtained from the simplification of a real application: a law firm composed by a number of lawyers manages various customers concurrently. Lawyers operate using a range of devices: desktop PCs, mobile phones, tablets, etc., which sometimes need to work off-line due to lack of connectivity. The two key requirements for this application are: (1) Data must be always available, even when devices are off-line; (2) Data must be synchronized: if a change is made, this should be pushed to all the other devices as soon as a connection is available.

To develop this application, we can start from a framework such as Spring that provides a Java-based back-end and a standard web client implementation

[4] From now on we will omit the fact that server-side development includes also a browser client, when this is clear from the context.

using an MVC pattern. More in detail, the development process is similar to the following:

1. Develop the server-side model using a dedicated tool (this is typically a domain specific language, or a graphical tool).
2. Develop controller and view for the browser-based client: source code for these is normally generated automatically from the client-side model description, and the developers only need to introduce changes for the specific application.
3. Develop server-side views (and, when needed, a controller) for REST (or SOAP) API. These APIs are used by the native mobile clients that need to interact with the "main" server.

This process enables the creation of the web-based application that lawyers can use from their desk. However, in many cases lawyers need to work from remote locations (e.g. court, at clients' residence, etc.), and in many cases connectivity is not available (e.g. in tribunals). Suppose now we want to develop a native Android client to be used in these circumstances. Effectively, this results in the development of a new, separate application that interacts with the previous one by means of REST (or SOAP) APIs. The typical steps would be:

1. Develop the client-side model and persistence model. Android has currently no support for automated model and persistence generation. The implementation is completely left to developers which have to choose among various solutions including the Google App engine [5] and a Sqlite DB.
2. Develop the client-side view and controller (i.e. the user interface). As above, these may be partially generated automatically from the model. item Implement the client-side REST (or SOAP) API to synchronize with the "main" server.

Notice that all the issues associated with synchronization (e.g. pushing local changes or fetching new data) need to be resolved by the developer on the client-side API.

In addition to mobile phones running Android, a number of lawyers have access to tablets such as the Apple iPad. The implementation of a native iOS application would follow the same steps above using the iOS SDK, but for iOS clients there is another possibility that avoids the need of placing the iOS mobile client on Apple App Store: developing a iOS web app using the Javascript extensions of Safari mobile[5]. This can be considered as a Javascript-based mobile application in which the browser itself act as a sandbox for the application. The interaction with the "main" server in this case requires a server-side MANIFEST file listing all the resources required to execute the application. The browser drives the application life cycle with a sequence of events and allows the Javascript code to read and write external resources such as private property files or databases.

[5] This is the solution currently adopted by `http://www.ft.com`. Web apps can also be implemented using Chrome extensions, see below.

In this particular case, the whole client implementation is embedded in server-side views or files which are loaded by Safari mobile when the application bootstraps. More in detail iOS web apps require to:

1. Develop a Javascript client-side model.
2. Develop all the views and the Javascript controller.
3. Implement a Javacript client for the REST API. (The implementation of a Chrome web extension follows an identical pattern.)

Both the native Android/iOS development and the web app development presented above show that the current implementation strategies lead to a substantial duplication of code that is clearly interdependent, and to a number of inefficiencies:

- The MVC pattern is repeated in all clients.
- A change in the "main" (server-side) model requires the immediate change of all the clients' models, making the code hard to maintain. As an example, suppose that there is a change in the customers' records, e.g. by introducing an additional field in a table. The models of all the mobile applications need to be modified accordingly before being able to interact with the server again.
- For each mobile client, developers have to implement manually all the aspect of synchronization, for each possible platform.
- Testing the application (as a whole) is problematic: for instance, fixtures need to be developed *for each* possible client and for the "main" server.

3 Symmetric Synchronized MVC

In this section we describe SSMVC (Symmetric Synchronized MVC), our proposed extension of the MVC pattern.

MVC-based frameworks ask developers to implement model classes representing persistent stateful objects. Starting from those classes each framework is capable of creating (semi-)automatically the database schema, the REST API, detail and list views on each model. If the original model changes, the framework is also capable of (semi-)automatically updating the code [6]. However, as in the development case, this update operation is repeated on the server and on each client separately.

3.1 Symmetric MVC

One key point of our idea is to consider web-based and mobile clients as a *single application*. Following this idea the same framework that is generating the server-side database, views and controllers can also generate all the client code. Similarly changes in one of the models can be (semi-)automatically and simultaneously updated by the framework itself.

Notice that traditional client applications use API to synchronize with the server. As we stated in Section 2, both server and client endpoints have to

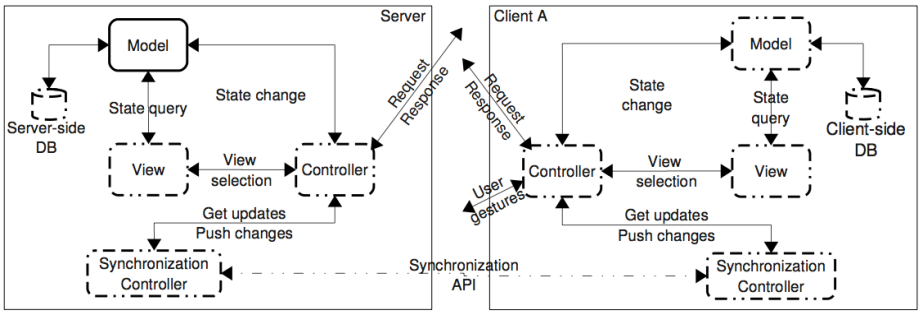

Fig. 1. Symmetric Synchronized MVC

be implemented. However, we argue that both of them can be automatically generated starting from the model.

Figure 1 depicts the SSMVC process, where the model in bold is provided by the user, and all the elements in dashed boxes are (semi-automatically) generated. More in detail, the generation process needs to:

1. extract the client-side model from the server model.
2. generate for each model the database schema and the platform-dependent persistence code (e.g. all the code containing the query to create, save and change models on the database).
3. generate on demand basic views and controller (e.g. a detail view on a model instance or a list view listing all the instances of a certain models).
4. configure a set of synchronization API on the server and their client counterpart to load data from the server or to push changes from the client.

Notice that most of these steps are *already* performed by existing MVC frameworks, but only for a single platform. Table 1 describes which part of the application are automatically generated by applying the MVC pattern to various server and mobile platform: the same steps can be repeated for SSMVC starting from a *single* model.

`Django-jom` is our initial prototype implementation of the SSMVC pattern for the Django framework. A detailed description of this implementation can be found at `https://github.com/msama/django-jom/wiki`. `Django-jom` stands for Django Javascript Object Models. The implementation is a Python Django component that automatically generates all the Javascript code necessary to export existing Django models for the server into Javascript objects for clients. Developers need to specify a descriptor indicating, for each model that they want to be exported, some basic properties, including which fields should be exported and how. Additionally, developers can implement a skeleton with additional prototypes that the Javascript objects should have at runtime. At this stage of the implementation `Django-jom` only handles the Symmetric behaviour. The Synchronization strategy described in the next section is currently being implemented.

Platform	Description
Server	Model: Manually implemented at abstract level. Persistence handled by the framework. View: Partially generated by the framework. HTML and CSS and custom views are manually added. Controller: Generated by the framework. Developer can add custom ones.
Web apps (Chrome/iOS)	Model: Javascript model and database connection are automatically generated. View: Partially generated by the framework. HTML and CSS and custom views are manually added. Controller: Default Javascript controllers are generated by the framework. Developer can add custom ones.
Native (iOS/Android)	Model: SQLite and core data are automatically generated as well as model classes. View: Automatic XML generation (Android); limited by XCode integration (iOS) Controller: Default controllers are generated by the framework. Developer can add custom ones.

Table 1. Automatic code generation in SSMVC

3.2 Synchronized MVC

Synchronizing client-server applications when clients can work both on-line and off-line is non-trivial. Traditional REST (or SOAP) API are designed to support the transfer of data when the client is on-line but they have no support for re-synchronizing clients operating off-line.

Consider for instance two lawyers updating the same data, one online and one off-line. The lawyer operating off-line will change data in the local database but will not be able to push changes to the server. When the lawyer returns on-line, the mobile client has to:

1. trace local changes which have not been pushed
2. pull an update from the server as soon as a connection is re-established
3. resolve possible conflicts on the locally modified data
4. resolve the conflict with a given policy (e.g. ask the user, merge or discard the changes)
5. push the update to the server
6. mark all the local changes as pushed to the server

None of the above features is automatically provided by the existing frameworks. Currently, implementations are left to developers which have to implement this mechanism for each model.

Our proposed Synchronized MVC looks at this problem from another perspective. Modern groupware implement a number of strategies to handle disconnection and reconnection of clients. For instance, the DiscoTech toolkit [4] provides an API to manage *event queues*, both at the client and at the server

side. The specific instantiation of a strategy is application-dependent, but the toolkit provides a parametric framework that can be adapted by the application developer to manage the possible events of a particular application. Notice that given the model and given a framework supporting the synchronization issues, all the system code could be automatically generated. Figure 2 provides a high-level overview of the toolkit; we refer to [4] for additional details.

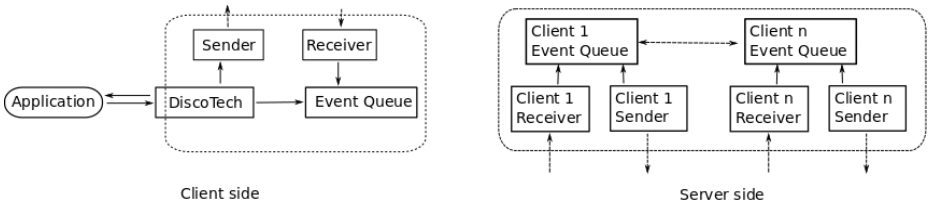

Fig. 2. Overview of the DiscoTech toolkit (from [4])

Conflict resolution is one of the most important aspect in terms of usability and user experience. Various systems could implement different policies according to their needs. For instance in our example we could apply a policy in which junior lawyers cannot override changes made by senior lawyers. The definition of such conflict resolution policies is out of scope for this paper but we remark that the SSMVC can accommodate these policies.

For the sake of completeness, there is an additional issue which needs to be addressed: changes in the model while the system is being used. This is a common issue in every data-driven application. Existing system such as iOS web applications employ a mechanism to check if the database version has changed and to trigger an update script. In SSMVC we can apply a more efficient solution. If the model (and, therefore, the database) changes, we can add the appropriate event to the queue of events, thus distributing it to all the clients as if it were a data change.

4 Related work

The MVC pattern is broadly used in industry and most of the existing web and mobile frameworks adopt it as base for the development. There has been some work in the extension of the MVC pattern. For instance Huang and Zhang [7] add an extra layer using XML and XSL. Their approach goes in the direction of using the MVC pattern for the server side application but does not include mobile clients. Other extensions include Flexible Web-Application Partitioning [8], targeting broswer-based web applications and MVC RIA (Rich Internet Applications) [9]. The latter work suggests splitting the model in client and server models, adopting browser plug-ins for interoperability of code, and mentions the use of a client-side persistence layer. However, the issue of synchronization is not addressed in this work.

More in general, the concept of synchronizing client applications, allowing users to access their data from multiple platforms, is becoming increasingly popular in industry. Server-side Chrome extensions [10] support client synchronization. Off-line changes are synchronized and distributed to all the clients employed by the same user (e.g. see the synchronization mechanism of Google Calendar for web browsers, phones, and tablets). However, there is no coherent solution: we found examples in which clients cannot make modifications if they bootstrap off-line, even when they go back on-line. Our pattern associates synchronization with concepts from CSCW, allowing multi-user concurrent pushes.

Other works exists in literature for mobile client synchronization. Shun at al. [11] propose a cache-like synchronization system. They introduce an intermediate state in which a client operating off-line has to re-synchronize before operating on-line. They also propose a coordination algorithm. The synchronization part of SSMVC is similar in spirit, but instead of proposing a new synchronization algorithm we suggest to use a Git-like protocol, and we also look at the global architecture of an application.

There is a substantial body of work on collaborative applications, see for instance [12] and references therein. The SSMVC pattern makes use of ideas from CSCW to address the issue of synchronization among multiple clients. Overall, however, our focus is different: the applications for which we suggest the use of SSMVC are not necessarily groupware applications (in the sense that multiple users collaborate to achieve a common goal), but could be applications in which a single user access data from a range of devices, not always connected. Additionally, we address the issue of *code dependency* and *code generation* for deployments on heterogeneous platforms, and to the best of our knowledge this is an issue that has not been addressed by the CSCW community.

5 Conclusion

The idea we propose here is a first step in the direction of considering backends, mobile applications and synchronization of user data as components of a *single* distributed and coordinated data-driven system. In this sense the SSMVC pattern that we propose in this paper aims at aiding web and mobile application developers by introducing a "design once and deploy everywhere" principle, with the additional benefits of avoiding code duplication, improving efficiency, testability, and maintainability. We achieve this by exploiting the *dependencies* between the various components of the overall system.

We consider our MVC extension *symmetric* because the same architecture is replicated both on the server and on each client platform. We consider our MVC extension *synchronized* because it gives developers a way to handle on-line and off-line concurrent changes by exploiting synchronization strategies developed for groupware applications that can be automatically generated.

We are currently developing a tool implementing model-to-model transformations to support SSMVC. A preliminary prototype implementing the Symmetric MVC pattern is available at `https://github.com/msama/django-jom/wiki`; in

this prototype we generate Javascript code automatically from a server model for the Python-based Django framework. We believe that the SSMVC pattern could substantially improve the current approaches to web system development. We hope that this initial submission will provide feedback and comments on the feasibility of our idea, and suggestions for improvements to the process as a whole and to its single components.

References

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Professional (1995)
2. Django: `https://www.djangoproject.com/` Accessed: 26/06/2012.
3. Spring: `http://www.springsource.org/` Accessed: 26/06/2012.
4. Roy, B., Graham, T.C.N., Gutwin, C.: Discotech: a plug-in toolkit to improve handling of disconnection and reconnection in real-time groupware. In Poltrock, S.E., Simone, C., Grudin, J., Mark, G., Riedl, J., eds.: CSCW, ACM (2012) 1287–1296
5. Google App Engine: `http://developer.android.com/training/cloudsync/aesync.html` Accessed: 29/06/2012.
6. South: `http://south.readthedocs.org/` Accessed: 26/06/2012.
7. Huang, S., Zhang, H.: Research on improved MVC design pattern based on struts and xsl. In: Information Science and Engineering, 2008. ISISE '08. International Symposium on. Volume 1. (dec 2008) 451 –455
8. Leff, A., Rayfield, J.: Web-application development using the model/view/controller design pattern. In: IEEE EDOC'01, IEEE (2001) 118–127
9. Morales-Chaparro, R., Linaje, M., Preciado, J., Sánchez-Figueroa, F.: Mvc web design patterns and rich internet applications. Proceedings of the Jornadas de Ingeniería del Software y Bases de Datos (2007)
10. Chrome web extension: `http://code.google.com/chrome/extensions/storage.html#apiReference` Accessed: 26/06/2012.
11. Shun-Yan, W., Luo, Z., Yong-Liang, C.: A data synchronization mechanism for cache on mobile client. In: WiCOM06. (2006) 1 –5
12. Avgeriou, P., Tandler, P.: Architectural patterns for collaborative applications. Int. J. Comput. Appl. Technol. **25**(2/3) (February 2006) 86–101