Models and Model Transformations Within Web Applications

Sergejs Kozlovics(✉)

Institute of Mathematics and Computer Science,
University of Latvia (Riga, Latvia), Raina blvd. 29, Riga 1459, Latvia
sergejs.kozlovics@lumii.lv

Abstract. Unlike traditional single-user desktop applications, web applications have separated memory and computational resources (the client and the server side) and have to deal with multiple user accounts. This complicates the development process. Is there some approach of creating web applications without thinking about web-specific aspects, as if we are developing stand-alone desktop applications? We say, "yes", and that is where models and model transformations come in handy. The proposed model-driven approach simplifies the development of web applications and makes it possible to use a single code base for deploying both desktop and web-based versions of the software.

Keywords: Models · Model transformations · Web applications

1 Introduction

In 2001, Model-Driven Architecture (MDA) was considered a promising approach for software development [30]. Indeed, models are a universal tool for system and data modeling; they can be used at different levels of abstraction — from domain-specific languages familiar to domain experts to platform-specific aspects and neat implementation details. Automated transformations between such models could replace traditional compilers. Although MDA and its further developments like Model-Driven Engineering (MDE), Model-Driven Software Development (MDSD), and other MD* have known success stories, some experts consider that the model-driven approach (in general) has "missed the boat" [13,14,18,29]. Although models are still in honor within academic researchers, most practitioners continue to use traditional technologies (relational databases and popular programming languages such as Java, C#, etc.; not models and transformation languages) due to lack of stable, production-ready model-driven infrastructure.

Nevertheless, another "boat", by which models could go, appears on the horizon — web-based software development. Currently, the majority of web application is developed using well-known server-side technologies (PHP, SQL and no-SQL databases, etc.), web protocols, and the client-side HTML+CSS+JavaScript stack. While creating a web-based application, the developer has to think about

© Springer International Publishing Switzerland 2016
G. Arnicans et al. (Eds.): DB&IS 2016, CCIS 615, pp. 53–67, 2016.
DOI: 10.1007/978-3-319-40180-5_4

server-side code, the client-side code, the communication issues, and the user-specific aspects (authentication and access control). This complicates the development of web applications, since network-specific issues have to be considered *in addition* to the primary functionality of the software. Moreover, it may be hard to choose where the particular computation-intensive code has to be executed — at the server side or at the client side. For instance, we faced this dilemma when considering layout computation for graph-like diagrams.

In this paper we show that if we “resurrect” models and take them on board, web-based applications can be developed much easier. In our approach, models are used as a memory (Random Access Memory, RAM) analog, which is automatically synchronized between the client and the server, thus, making network communication transparent. We also show how models help to manage the resources (processor and memory) automatically and transparently between multiple users, who can use the application simultaneously. Thus, the developers can just assume a single user PC as a target.

The paper is structured as follows. First, we describe our previous approach of using models and model transformations in classical desktop applications (Sect. 2). Then we adapt it to meet the requirements of the web (Sect. 3). We continue by providing solutions for certain issues arising when moving models to the web. The “Related Work” section lists several alternative approaches for developing web applications. It is followed by the conclusion, which presents our experimental results and points to further research directions.

2 Traditional Approach

In 2008, we have proposed a domain-specific desktop tool building approach called the Transformation-Driven Architecture, TDA [7,22]. TDA has been successfully used to implement several domain-specific tools such as ProMod, OWLGrEd, and VisiQuer [4–6]. The main concepts of TDA are model transformations, engines, and interface metamodels (Fig. 1). Model transformations are

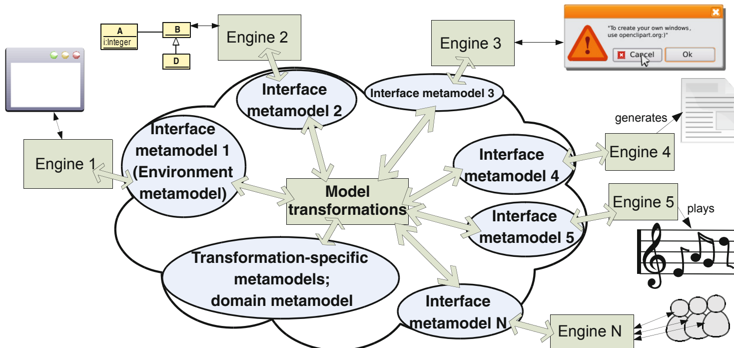

Fig. 1. The outline view on the Transformation-Driven Architecture, TDA.

used to implement business logic. Unlike MDA, TDA uses model transformations at runtime. Engines are pluggable modules that provide certain auxiliary functionality (such as services and graphical presentations) for transformations. While transformations are usually written in a platform-independent way using model transformation languages or traditional programming languages, engines are usually implemented using platform-specific libraries and technologies. There may be multiple variations of the same engine for different platforms (different operating systems).

Model transformations communicate with engines via instances of interface metamodels. There are special classes called events and commands (subclasses of the Event class and the Command class). When a transformation needs to call some engine, it creates a corresponding command instance, sets its attribute values and creates necessary links to specify the command arguments and the context. Then the command instance is linked to the submitter object in the model, which is treated as a request to execute that command. Engines, in their turn, are able to emit events, when certain actions (e.g., user clicks) occur. In order to catch an event, some event-handling transformation has to be registered as an event listener. Events are emitted in the same way as commands (e.g., an event object is created and linked to the submitter). On the one hand, such event/command mechanism keeps transformations away from technical issues of calling different engines, which may be written in different programming languages. On the other hand, engines can be written in traditional languages, without the need to know how to call particular model transformation, which may be written in some specific transformation language or in some ordinary high-level language. All calls between the engines and transformations are performed automatically when corresponding links to the submitter are created. TDA has different adapters for different programming and transformation languages.

Models are stored in a model repository (in-memory repository, in most cases). Engines and transformations access models via a common API (we call it Repository Access API, RAAPI[1]) implemented for various programming languages and platforms. Certain RAAPI wrappers have been developed to provide query-based repository access (e.g., the lQuery language [24]). There are also some code generators that produce C++/Java classes that can be used to access objects stored in the model repository as if they were C++/Java objects. Thus, RAAPI (or one of its wrappers) is the only API a particular transformation or engine has to be aware of to be able to work within TDA.

3 Bringing Models to the Web

Now we show how the above-mentioned architecture for desktop tools can be scaled for web-based applications. The first approximation is as follows:

[1] http://tda.lumii.lv/raapi.html.

– Transformations, which implement business logic, remain intact. They are implemented using traditional high-level or transformation languages and executed at the server side.
– Engines, which mainly implement graphical presentations, are executed in the client browser. Engines must be re-written in JavaScript (or other language that translates to JavaScript), utilizing HTML and CSS, to provide a neat user experience without the need to install support for non-JavaScript languages. We may think of web engines as engine variations for the web platform. The interface metamodels of web engines remain intact, thus, we do not need to re-write transformations (but a TDA adapter for web-based engines is needed).
– Communication between engines and transformations now has to be implemented using network technologies.

Rewriting engines in the JavaScript+HTML+CSS stack as well as introducing a web-server to deliver the code of web engines to the client browser is a technical straight-forward process (once it is done, we need to maintain just the web-based engine, since it can be used also for desktop tools). But ensuring the communication between engines and transformations over the network involves more complex issues, including:

– bi-directional communication (we have to communicate in both directions by means of commands and events);
– asynchronous execution of commands and events;
– accessing the model repository from the server and from the client.

The next approximation is to get rid off the traditions and allow transformations to be executed right in the client browser and allow non-interactive engines to run at the server side. Thus, certain transformations and engines can be launched without the round-trip delay between the client and the server.

To complete the picture, we introduce multiple users. This includes user authentication and sharing server resources (processor and memory) between multiple users, with the potential scaling in mind.

The issues we have just mentioned are addressed in the next section.

Notice that among these issues, every web application has to consider potential security risks. We assume that best practice recommendations for preventing typical attacks are always kept in mind (e.g., escaping of HTML strings, using secure HTTPS/WebSocket connections, session checks, etc.) [20]. Security issues are mostly technical and are not considered in this paper.

4 Dealing with the Issues

4.1 Bi-Directional Communication Issues

The traditional HTTP protocol, designed in 1992, was developed to be a client-server stateless protocol: a client initiates a request and waits for the server to respond; each next request is treated as independent, since no state is stored at the protocol level. To use HTTP for bi-directional communication between

transformations and engines, where multiple users may be working with different models, we need some means for the server to initiate the communication (in order transformations running on the server side could send commands to engines running at the client side). A session identifier is also required for each authenticated user. These are well-known issues. Widely used solutions for bi-directional HTTP include long polling, when the client asks the server for commands at certain intervals, and COMET, where the client initiates a request, but the server delays the response until some command has to be sent to the client [15]. The traditional way to identify the session is to use the JSESSIONID cookie along with the list of active sessions at the server. Certain libraries, such as DWR[2], are able to factor out these technical issues. Still, in order to pass events and commands over the network, we need to serialize and deserialize them.

The WebSockets protocol, standardized in 2011 (drafts from 2010), is intended for high-speed bi-directional communication [16,19]. The protocol does not perform a handshake each time a message is sent. Moreover, message encoding overhead is minimal (compared to heavyweight HTTP headers). Currently, all recent versions of popular web-browsers support web sockets. Still, if we go for web sockets, we need some TDA event/command serialization or synchronization solution. We discuss it in Sect. 4.3.

To ensure the correct execution of commands and event handling, we need two additional TDA components, which are present regardless of the particular technique used for network communication. They are the web engine adapter and the client-side command manager[3].

Web engine adapter. When a transformation creates a command and stores it in the model repository, TDA calls the corresponding engine via a specific adapter. There are different adapters for different types of engines, usually, depending on the programming language or calling conventions (DLL, Java class, .NET assembly, etc.). We can assume that web engines have to be called via a special TDA "JavaScript" adapter. However, unlike traditional engine adapters, which work locally, this adapter starts the web-server (if it has not been done before), serializes the command, and sends it to the client browser.

Client-side command manager. Each TDA-based tool has Environment Engine, which is responsible for creating the main application window and attaching/detaching child windows. In case of web-based TDA, Environment Engine occupies one browser window (or tab), while other windows are attached as embedded frames (iframes) by means of some windowing library such as jQueryUI[4] or Dojo[5]. Each TDA engine has some function for processing commands. For desktop-based TDA tools, TDA takes care of calling the appropriate function for the given command, since the engines are attached locally. However, for web-based applications, engines are at the client side. Thus, we need

[2] http://directwebremoting.org/.
[3] we do not need a manager for events, see Sect. 4.3.
[4] http://jqueryui.com/.
[5] http://dojotoolkit.org/.

some client-side command manager, which determines the correct engine and its iframe, and passes the command to that frame. For web-based TDA, the manager can be a part of Environment Engine. It listens for command messages from the server. Then, given a command object, the manager searches for a corresponding *Frame* object in the model repository (each presentation engine must have created such object). Since Environment Engine already maintains a map that associates *Frame* objects with iframes, the command manager can use this map to get the correct iframe and forward the command message to it.

4.2 Asynchronous Issues

The major arguments for introducing asynchronous calls between the client and the server are as follows:

- When some engine (running in the browser) emits an event, some event-handling transformation is called at the server side. In order not to freeze the browser (taking into a consideration the network latency and the event handling transformation execution time), event handling should be asynchronous.
- When some transformation at the server side creates a command for some presentation engine, the engine usually needs to repaint some GUI elements. There is only one JavaScript thread in the browser; the thread is common to all engines, thus, command processing should not block other engines. Moreover, a separate GUI thread (which is the JavaScript browser thread is our case) is a de facto best practice standard (otherwise, deadlocks are inevitable) [10]. Since there is only one GUI thread, all GUI operations must be enqueued (as it is in the case of JavaScript operations), and thus, they cannot be synchronous[6].
- When some transformation at the server side creates a command for some engine, it must not block the server. Then other users can use the server resources, while the asynchronous command is being executed at the first user's browser.

Based on this, we require all events and commands of web-based engines to be asynchronous[7]. Thus, when a command or an event is being submitted, the caller thread is not blocked. If a callback is needed, an engine can emit an event, when it finishes processing the command, and a transformation can issue a command after the event has been handled. In the latter case, if the engine needs to repaint its presentation, it can use some optimistic prediction technique to visualize the expected state before the transformation finishes (the state can be adjusted later, if needed).

To support asynchronous communication, we introduce the *AsyncCommand* class in the metamodel (all events have been already asynchronous in TDA).

[6] Similar approach is used in traditional GUI libraries, such as Java Swing (the function SwingUtilities.invokeLater), JavaFX (the function Platform.runLater), Borland/Embarcadero VCL (the Synchronize function), etc.

[7] Any bi-directional communication technique mentioned in Sect. 4.1 can be used asynchronously.

Since all GUI commands of existing presentation engines are asynchronous, we can just make them subclasses of *AsyncCommand*. The TDA event/command mechanism now checks whether the given command is asynchronous. If yes, the command is forwarded to the corresponding engine. For desktop-based tools, TDA supports synchronous command calls as well (thus, when the transformation emits a command, the control is returned only when the command execution has finished).

4.3 Accessing the Model Repository from the Server and from the Client

Since model transformations use the model repository intensively, it is reasonable to run the repository at the server side. Before sending commands (with their context) to engines, commands are serialized. We may expect that the serialization should contain all the necessary information for the engine. However, engines may need to access objects that are not directly linked to the commands (not in the context). The full context serialization (or the full repository serialization in the worst case) is unreasonable, if the engine has to visualize just some of the objects. Also, when an event occurs, the engine needs to store it in the repository. Thus, some means to access the repository from the client side (engine side) is needed. There are two approaches:

- Provide some client-side query mechanism, while keeping the repository at the server side.
- Synchronize the repository between the client and the server

The first approach requires some query language. Our first approximation is to provide functions such as *findObjects*, *loadObjects*, *storeObjects*, and *deleteObjects*. The arguments and the result are in the JSON syntax (see Fig. 2).

The second (synchronization) approach requires some means to synchronize the server-side repository with some client-side data structure, containing the same model. The synchronization can be done in several ways:

- By means of bi-directional HTTP implementations (e.g., COMET). Since HTTP connection has to be established on each message (and this involves certain delay), it is reasonable to synchronize models in batch mode. For instance, repository write operations can be recorded at the server side while a transformation is being executed. When a command is being issued, the collected write operations are serialized and sent to the client browser. Likewise, while an engine is performing some operations, all model changes are recorded and then sent back to the server on events.
- By means of web sockets. The benefits of web sockets are:
  - the connection has to be established only once;
  - keeping the connection alive involves almost no overhead;
  - the connection is asynchronous, but the order of messages is preserved;
  - data do not need to be serialized, since binary communication is possible.

```
{
  isKindOf: "Person",
  name: "John"
}
```

(a)

```
{
  reference: 1001,
  class: "Person",
  name: "John",
  children: [ {
    reference: 1002
  }, {
    reference: 1003
  }]
}
```

(b)

Fig. 2. (a) A JSON object representing a query for *findObjects* for finding a Person with the given name. (b) A possible result of that query. Links are encoded as JSON arrays. Two special attributes, *reference* and *class*, specify the object identifier in the model repository and the class name, respectively. The *loadObjects* function can be used then to get attribute values for objects 1002 and 1003.

Thus, all repository write operations can be sent to a web socket right away, without introducing a special buffer for batch processing.

– By means of existing infrastructures, which provide automatic data synchronization. For instance, we can use the Meteor[8] infrastructure for that. Meteor stores data on the server side in a MongoDB and implements a common query language for both the server and the client, while keeping data synchronization transparent. While MongoDB is optimized for efficient queries, it has slow write operations.

Unfortunately, the client-side query mechanism as well as the HTTP synchronization requires data serialization/deserialization to/from JSON syntax. It proved to be slow in our experiments, where it may take around 2 seconds to serialize/deserialize graph diagrams of moderate size (around 100 elements), including network delay. The Meteor/MongoDB approach is optimized for efficient queries, but it has slow write operations (around 10000 write operations per second on a 3.4 GHz i7 processor, including optimizations). Our experiments show that existing transformations use write operations quite intensively (see Sect. 4.5), thus, only a small number of users can be connected to a web application without exceeding the processor power limits. The only feasible solution (from the above) is to use web sockets. In fact, web sockets allow data synchronization to be performed in parallel with computation. Still, we need some means to represent data at the client side. Binary JavaScript objects stored according to the syntax from Fig. 2 can be used for that. If we use binary web sockets, expensive JSON serialization can be replaced with lightweight object creations or attribute updates. While such JavaScript objects can be "touched" directly for read access, write access requires special functions ("setters"), since we need to listen to the changes to be able to synchronize them back to the server. In the example from Fig. 2, the object would be augmented with the functions *setName* and *setChildren*.

[8] https://www.meteor.com.

The server-side repository automatically takes care of launching engines and transformations, when command and event objects are put into the repository and connected to the submitter object. The client-side repository replica, however, needs some adjustments. In case of Meteor, we can introduce a client-side listener, which listens to new command objects and passes them to the client-side command manager. In case of HTTP/web sockets, when a message is received at the client, the message is analyzed. If the message contains a repository write operation for connecting a command object to the submitter, the client forwards the command to the client-side command manager.

With events, the process is much simpler. The client just creates an event and links it to the submitter at the client side. When the event reaches the server (during synchronization), the server-side repository will process it as usual.

We have already mentioned that code generators can be used to generate wrappers for repository classes in different programming languages (C++, Java, etc.) in the traditional TDA. They can still be used for server-side code, but synchronized client-side JavaScript objects are already native JavaScript objects. Thus, regardless of the programming language (and the server or client side), developers of transformations and engines may treat repository objects as native OOP-objects in RAM. Moreover, since the synchronization between the client and the server is automatic, the developers can assume they are writing applications for a single PC. We believe that this is an important benefit that models can bring. Models are like lens, beyond which network aspects can be hidden. In addition, function calls can also be implemented in a way native to the particular programming language by providing glue code for commands and events in the repository.

4.4 Server-Side Engines and Client-Side Transformations

Most TDA engines are graphical and, in case of web applications, are executed in the browser. Still, some engines can perform certain computation without the need to visualize anything, but requiring server resources (e.g., intensive computations are inefficient, if running as browser scripts). On the other hand, it may be reasonable to develop certain model transformations in JavaScript to be executed directly in the browser (e.g., small GUI event handlers, which may need just to adjust the presentation). Thus, server-side engines and client-side transformations are needed.

Technically, if an engine does not have any own graphical presentation, it can be executed at the server side by means of existing TDA event/command mechanism. When a command for a not-web-based engine is created, it is passed to some adapter at the server side, which executes the command.

For client-side event handlers the process is not that easy. We need a client-side event manager, which monitors write operations that are sent from the client to the server. If an event is being sent, the client-side event manager gets the associated event listener (its name is stored in the repository). If the corresponding event listener is written in JavaScript and has to be run at the client side, the event manager executes that listener right away. The event may still be

posted to the server, where the corresponding client-side JavaScript adapter will be searched. Since no such adapter exists (or, we may create a fictive adapter, which ignores all its events), the event will not be executed at the server side. Another option is just not to post the event to the server, but then the client and the server repositories would not be identical until the event object is deleted at the client side.

If all transformations are client-side transformations (called as event handlers), then the whole application can be run within a web browser. To persist the model, either a server-side repository, or a third-party cloud storage can be used. For instance, DropBox[9] and OneDrive[10] files can be used to store the serialized models (public APIs are available). If no server-side transformations and engines are present, then no model synchronization is needed. Models can be loaded from the cloud, when the web application is loaded, and saved on exit (or on regular basis).

4.5 Multi-user Issues

For authentication we can use a traditional login/password approach, or delegate the authentication to third parties (such as Google or Facebook). Web-based Environment Engine implement the client-side authentication, while the server process performs necessary checks and marks the given user (associated with the given HTTP session) as logged in. For Meteor-based variant, we can use Meteor authentication with plugins.

For traditional desktop-based TDA we used our proprietary repositories (MII_REP/OUR and JR) as well as ECore files to store models [8,31,33]. Since there may be multiple users accessing their models at the same time, for web-based TDA we can use a traditional or no-SQL database as well. Databases implement all necessary services, such as disk cache, optimized search, support for multiple threads/processes, etc., which are essential in a multi-user environment.

It is reasonable that one server process is dedicated to the web-server. Another one may be dedicated for the database. Depending on the number of processor cores at the server, we can create N worker processes, which can be used to execute server-side transformations and engines. Ideally, N would be equal to the number of processor cores minus 2, since 2 processes are occupied by the web server and the database. Each server-side transformation or engine call is enqueued and then processed by one of the N "workers".

Another solution is to create N workers, where each worker has its own in-memory database (instead of a common single database). Thus, we do not need a dedicated database process, and if the number of users does not exceed N, they can use the server in parallel. When the $N+1$-th user comes in, the in-memory database of the user, who was idle for the longest time, is flushed to the disk, and the repository of the new user is loaded in that place.

---

9 https://www.dropbox.com.

10 https://onedrive.live.com.

In case of some runtime exception in a server-side transformation, the corresponding worker process can be terminated (and a new "healthy" process can be launched for further transformations). In case of a common database, the previous repository state (before the error) has to be restored. In case of in-memory database, no actions are required (since repository flushing to disk is performed only after successful execution of server-side transformations).

For a server having 8 GB RAM, we can assume 4 GB are free. Taking into a consideration our experience with desktop-based TDA, the 100 MB upper bound for each in-memory repository seems to be reasonable. Thus, a single server can serve up to 40 users without swap. Based on our existing experience, we can assume that each transformation performs 1000 model operations on average[11]. Thus, we have 40000 model operations per 40 users, where 10000, could be write operations and 30000 read operations. Our in-memory repositories can deal with 40000 operations in a few milliseconds. Since repository actions are synchronized asynchronously at once, we just need to add the network delay (usually, 100–200 ms), thus the total time is below one second, which is considered adequate [26,27]. MongoDB (used by Meteor), in its turn, has slow write operations (10000 write operations could be executed in 1000 ms on a 3.40 GHz processor, if we use the batch mode). Thus, we can assume 10000 write operations take one second, while other 30000 read operations take another second, resulting in 2 s, which is less efficient and is at the bound of the "seamless" user experience [28].

Notice that the calculations above are at the full load of 40 users, who emit events each 1 or 2 s. In practice, transformations are called occasionally, thus, more users can be connected and using server resources without interference (still, for the in-memory solution, repository flushing/loading may be required). As a result, the number of simultaneously connected users may reach several hundreds. For thousands of users, we need to configure load-balancing between multiple servers. While the number of 10000 users is considered appropriate for existing operating systems (the C10K problem), serving millions of users (the C10M problem) requires bypassing the OS by using sophisticated techniques (and currently this is not our goal).

5 Related Work

In 2007, de Castro et al. presented an MDA-based approach for developing service-oriented web applications [17]. This approach has been applied using the AMMA tools and the ATL transformation language for modeling web applications [1,3,12]. A different, but also MDA-based approach was used in VisualWade[12] (currently obsolete). It was intended as a out-of-box product for generating PHP code for web applications from source models, which could be defined graphically with a few mouse clicks. While models were used at the development stage, traditional databases such as MySQL, PostreSQL, and ORACLE were

[11] These are transformations with non-intensive computation, as transformations in existing TDA-based tools.

[12] http://visualwade.software.informer.com/.

used at runtime. Other similar tools include WebRatio[13] (using Web Modeling Language, WebML[14]) and OpenUWE (using the ArgoUWE, an ArgoUML-base tool) followed by MagicUWE [11,21]. The WebSA approach also uses MDA, but explicitly focuses on the functional requirements of web applications [9]. WebTA is a transformation engine specifically designed to bring transformations into WebSA [25]. An extensive survey on different MDA-based approaches for web applications can be found in the article by Schwinger and Koch [32]. Still, all these approaches use models and transformations at development time. This differs from our approach, where models and transformations are executed at runtime. Currently, the MDA/MDE field is in the state of stagnation (especially, after the Bezivin "Why did MDE miss the boat?" talk in 2011 [13]). We believe that our web-based approach can give a new breath to models, thus new results in the field can appear.

While we use either a model repository or a database to store models, one can use third-party cloud storage for that (reasonable, when all transformations and engines are running at the client side). While we can use any file format for that, using spreadsheets (like Google spreadsheet[15] or Microsoft Excel online[16], which have JavaScript APIs) and the appropriate encoding (e.g., sheets are classes, rows are objects, and columns are attribute values), we also get a free tabular repository browser for debug purposes as a by-product.

The EASA Spreadsheet Deployment platform is an interesting approach to creating web application from Excel files [2]. The Excel file is treated as a source model, from which an out-of-box web application is obtained.

Google Apps Script[17] is a platform to developing web applications intended to be run in a web browser. Google provides a graphical form designer, where JavaScript code can be attached to user events (clicks). Since applications are being executed in a web browser, additional services are required to persist data (e.g., to store user projects). Since Google Apps Script integrates with Google services, Google Drive can be used as a storage device. Still, if we need certain server-side computation or access to some proprietary database, the integration does not work; we need to create a web-service or some API for that. Thus, the system becomes heterogeneous. In contrast, our TDA-based approach provides persistency automatically, since all the models are saved in a repository. Both the server and the client use the same model (and they do not need to be aware of model synchronization). The TDA event/command mechanism is a unified way to call transformations (functions) or web-services regardless of their location and particular protocols used.

RollApp[18] is an interesting solution for bringing existing desktop applications to the web. It builds a Linux-based virtual environment, where the content of the

[13] http://www.webratio.com/.
[14] http://www.webml.org/.
[15] https://apps.google.com/products/sheets/.
[16] https://office.live.com/start/Excel.aspx.
[17] https://www.google.com/script/start/.
[18] https://www.rollapp.com/.

application window is sent to the browser (technically, this can be implemented easily, since X Window System already implements that feature). File dialogs are redirected to the cloud storage. Since not all programs are supported and mainly they are open source, we can assume that certain minor modifications of code are required. RollApp is a paid subscription. A a free plan, where the changes could not be saved, is also available. Although this solution provides a universal way of bringing desktop applications to the web, it requires much more server resources (processor and memory for creating a virtual environment) than creating web applications by means of traditional web technologies, where resources can be shared among multiple users more efficiently.

The m-Power[19] platform branch can be traced back to 1983. The goal is to build a web-interface for legacy applications. The platform uses traditional databases and Java for the resulting applications, and the process is not model-driven. We believe that models help the developers think at a higher level of abstraction, which is proposed by our approach. However, in case of legacy applications, which usually are not model-based, m-Power could be a better solution.

6 Conclusion

The paper provided a sketch of a TDA-based solution for developing web applications using models and model transformations at runtime. We have implemented a prototype, using the ECore repository for model storage. The prototype includes:

- the web-based Environment Engine (utilizing the Dojo toolkit for attaching child windows);
- a simple client-side query language (utilizing the JSON syntax mentioned in the paper) for accessing server-side repository from the client;
- some web-based engines, which have been developed or re-written in JavaScript.

A recent TDA-based DataGalaxy tool can be considered as approbation of the main ideas of the approach [23]. Since this tool has only web-based engines, they can be used in both desktop and web modes without change. Java transformations can also be used either as ordinary TDA transformations, or server-side transformations. Thus, the same code base can produce both desktop-based and web-based versions of DataGalaxy. This is the main strength of the approach.

We are working on developing web versions of engines used in our desktop-based ontology editor OWLGrEd, thus, OWLGrEd (as well as some other tools) can be launched in the web in the near future. The main benefit of our approach is that web based tools can be treated by developers as desktop-based tools. Another benefit comes from models. Models provide a universal platform-independent encoding for data as well as for operations. In the future this can lead to a high-level network-transparent RAM analog. We believe that the potential power of models will eventually reveal itself, if we start using models in the web.

[19] http://www.mrc-productivity.com.

Acknowledgments. The work has been supported by Latvian State Research programme (2014–2017) NexIT project No.1 'Technologies of ontologies, semantic web and security'.

References

1. ATL use case - modeling web applications. http://www.eclipse.org/atl/usecases/webapp.modeling/
2. EASA Spreadsheet Deployment. http://www.easasoftware.com/solutions/spreadsheet-deployment/
3. Allilaire, F., Idrissi, T.: ADT: Eclipse development tools for ATL. In: Proceedings of Second European Workshop on Model Driven Architecture (2004)
4. Barzdins, G., Liepins, E., Veilande, M., Zviedris, M.: Ontology enabled graphical database query tool for end-users. Frontiers in Artificial Intelligence and Applications, vol. 187, pp. 105–116. IOS Press (2008)
5. Barzdins, J., Barzdins, G., Cerans, K., Liepins, R., Sprogis, A.: OWLGrEd: a UML style graphical notation and editor for OWL 2. In: Proceedings of OWLED 2010 (2010)
6. Barzdins, J., Cerans, K., Kalnins, A., Grasmanis, M., Kozlovics, S., Lace, L., Liepins, R., Rencis, E., Sprogis, A., Zarins, A.: Domain specific languages for business process management: a case study. In: Proceedings of DSM 2009 Workshop of OOPSLA 2009, pp. 34–40, Florida, USA (2009)
7. Barzdins, J., Kozlovics, S., Rencis, E.: The Transformation-Driven Architecture. In: Proceedings of DSM 2008 Workshop of OOPSLA 2008, pp. 60–63, Nashville, Tennessee, USA (2008)
8. Barzdins, J., Barzdins, G., Balodis, R., Cerans, K., Kalnins, A., Opmanis, M., Podnieks, K.: Towards semantic Latvia. In: Vasileckas, O., Eder, J., Caplinskas, A. (eds.) Proceedings of Seventh International Baltic Conference on Databases and Information Systems, Communications, Lithuania, Vilnius pp. 203–218 (2006)
9. Beigbeder, S.M., Castro, C.C.: An MDA approach for the development of web applications. In: Koch, N., Fraternali, P., Wirsing, M. (eds.) ICWE 2004. LNCS, vol. 3140, pp. 300–305. Springer, Heidelberg (2004)
10. kgh blog: Multithreaded toolkits: A failed dream? Originally. https://community.oracle.com/people/kgh/blog/2004/10/19/multithreaded-toolkits-failed-dream, http://tecnologia.revistacocktel.com/multithreaded-toolkits-a-failed-dream/. Accessed 5 May 2016
11. Busch, M., Koch, N.: MagicUWE – a CASE tool plugin for modeling web applications. In: Gaedke, M., Grossniklaus, M., Díaz, O. (eds.) ICWE 2009. LNCS, vol. 5648, pp. 505–508. Springer, Heidelberg (2009)
12. Bzivin, J., Jouault, F., Rosenthal, P., Valduriez, P.: The AMMA platform support for modeling in the large and modeling in the small. Technical report, LINA, Universite de Nantes (2005)
13. Bzivin, J.: Why did MDE miss the boat? In: First International Workshop on Combined Object-Oriented Modeling and Programming (COOMP 2011) (2011)
14. Cabot, J.: Clarifying concepts: MBE vs MDE vs MDD vs MDA. Post at MOdeling LAnguages. http://modeling-languages.com/clarifying-concepts-mbe-vs-mde-vs-mdd-vs-mda/
15. Carbou, M.: Reverse Ajax, Part 1: Introduction to Comet. http://www.ibm.com/developerworks/library/wa-reverseajax1/index.html. Accessed 5 May 2016

16. Carbou, M.: Reverse Ajax, Part 2: WebSockets. http://www.ibm.com/developerworks/library/wa-reverseajax2/index.html. Accessed 5 May 2016
17. de Castro, V., Vara, J., Marcos, E.: Model transformation for service-oriented web applications development. In: Workshop Proceedings of 7th International Conference on Web Engineering, pp. 284–198 (2007)
18. Dubray, J.J.: Why did MDE miss the boat? (A summary). InfoQ News, 27 Oct 2011. http://www.infoq.com/news/2011/10/mde-missed-the-boat
19. IETF: The WebSocket protocol. RFC 6455. https://tools.ietf.org/html/rfc6455
20. Kern, C.: Securing the tangled web. Commun. ACM **57**(9), 38–47 (2014). http://dx.org/10.1145/2643134
21. Knapp, A., Koch, N., Moser, F., Zhang, G.: ArgoUWE: a CASE tool for web applications. In: Proceedings of the 1st International Workshop on Engineering Methods to Support Information Systems Evolution (EMSISE 2003) (2003)
22. Kozlovics, S., Barzdins, J.: The transformation-driven architecture for interactive systems. Autom. Control Comput. Sci. **47**(1/2013), 28–37 (2013). Allerton Press Inc
23. Kozlovics, S., Rucevskis, P.: Manipulating and visualizing data by means of data galaxies. Frontiers in Artificial Intelligence and Applications, vol. 270, pp. 85–98. IOS Press (2014)
24. Liepiņš, R.: Library for model querying: IQuery. In: Proceedings of the 12th Workshop on OCL and Textual Modelling, OCL 2012, pp. 31–36. ACM, New York (2012)
25. Meliá, S., Gómez, J., Serrano, J.L.: WebTE: MDA transformation engine for web applications. In: Baresi, L., Fraternali, P., Houben, G.-J. (eds.) ICWE 2007. LNCS, vol. 4607, pp. 491–495. Springer, Heidelberg (2007)
26. Miller, R.: Response time in man-computer conversational transactions. In: Proceedings of AFIPS Fall Joint Computer Conference, vol. 33, 267–277 (1968)
27. Nielsen, J.: Usability Engineering. Morgan Kaufmann, San Francisco (1993)
28. Nielsen, J.: Website response times (2010). https://www.nngroup.com/articles/website-response-times/
29. Object Management Group: MDA Success Stories. http://www.omg.org/mda/products_success.htm
30. Object Management Group: Model Driven Architecture. http://www.omg.org/mda/
31. Opmanis, M., Čerāns, K.: Multilevel data repository for ontological and meta-modeling. In: Databases and Information Systems VI - Selected Papers from the Ninth International Baltic Conference, DB&IS 2010 (2011)
32. Schwinger, W., Koch, N.: Web engineering: the discipline of systematic development of web applications (chap.) In: Modeling Web Applications, pp. 39–64. Wiley, Hoboken (2006)
33. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF Eclipse Modeling Framework, 2nd edn. Addison-Wesley, Reading (2008)