Interactive, Live Mashup Development through UI-Oriented Computing

Anis Nouri and Florian Daniel

University of Trento
Via Sommarive 9, I-38123, Povo (TN), Italy
anis.nouri-1@studenti.unitn.it,daniel@disi.unitn.it

Abstract. This paper proposes to approach the problem of developing mashups by exclusively focusing on the Surface Web, that is, the data and functionality accessible through common Web pages. Typically, mashups focus on the integration of resources accessible through the Deep Web, such as data feeds, Web services and Web APIs, that do not have own UIs – next to data extracted from Web pages. Yet, these resources can be wrapped with ad-doc UIs, suitably instrumented, and made accessible through the Surface Web. Doing so enables a UI-oriented computing paradigm that allows developers to implement mashups interactively and in a live fashion inside their Web browser, without having to program any line of code. The goal of this paper is to showcase UI-oriented computing in practice and to demonstrate its feasibility and potential.

Keywords: UI-oriented computing, iAPIs, mashups, integration

1 Introduction

The most notable technologies today to publish and access data and functionality over the Web are SOAP/WSDL Web services [2], RESTful Web services [12], RSS/Atom feeds, and static XML/JSON/CSV resources. Alternatively, data may be rendered in and scraped from HTML Web pages, for example, using tools like Dapper (`http://open.dapper.net`) or similar that publish extracted content again via any of the previous technologies. W3C widgets [4] or Java portlets [1] are technologies for the reuse of small, full-fledged applications that also provide for the reuse of user interfaces (UIs).

All these technologies (except the Web pages) are oriented toward programmers, and understanding the underlying abstractions and usage conventions requires significant software development expertise. This makes data integration a prerogative of skilled programmers, turns it into a complex and time-consuming endeavor (even for small integration scenarios), and prevents less skilled users from getting the best value out of the opportunities available on the Web.

UI-oriented computing (UIC [8]) takes a different perspective and starts from the UIs of applications we all – programmers and users – are accustomed with and that are free of developer-oriented abstractions. The research question UIC poses is if and, if yes, which of the conventional Web engineering tasks can be

achieved if we start from the UIs of applications, instead of from their APIs or services. The vision is to enable everybody to perform simple integration tasks directly inside their Web browser, for example, the integration of data extracted from different Web pages or the automation of repeated navigation actions.

In our prior work [9], we already investigated how to turn UIs into programmable artifacts and introduced the idea of *interactive APIs* (iAPIs), that is, APIs users can interact with via their graphical Web UIs. In [8], we then studied the specific case of *data integration* and described an end-to-end solution for UI-oriented computing consisting of an iAPI annotation format, a graphical editor for iAPI manipulation and integration, and a suitable runtime environment.

The *goal* of this paper is to showcase a more extensive case study (the one developed in the context of the Rapid Mashup Challenge) and to provide insights into the practical aspects of UI-oriented computing with the current prototype of our development and runtime environments. In particular, the goal is to highlight the benefits to both common users (interactive, live development without coding) and programmers (programmatic UIC via a dedicated JavaScript library).

Next (Sections 2 and 3), we introduce the concept and practice of UI-oriented computing along with its underlying runtime infrastructure. In Section 4, we then introduce the scenario we selected to approach the Rapid Mashup Challenge and how we prepared for the Challenge. In Section 5, we then describe the step-by-step development of the mashup scenario using the UI-oriented computing approach. We conclude the paper with a discussion of a set of works that are related to the proposal we push forward in this paper and a discussion of the findings, lessons learned and future works.

2 UI-Oriented Computing

The idea of UIC is to propose a new kind of "abstraction": no abstraction. The intuition is to turn UI elements into interactive artifacts that, besides their primary purpose in the page (e.g., rendering data), also serve to access a set of operations that can be performed on the artifacts (e.g., reusing data). Operations can be enacted either interactively, for example, by pointing and clicking elements, choosing options, dragging and dropping them, and similar – all interaction modalities that are native to UIs – or programmatically.

The core ingredient, interactive APIs, come as a binomial of a *microformat* for the annotation of HTML elements with data structures and operations and a *UIC engine* able to interpret the annotations and to run UI-oriented data integrations. The engine is implemented as a browser extension. A dedicated *iAPI editor* injects into the page *graphical controls* that allow the user to specify data integration logics interactively. The UIC engine maps them to a set of iAPI-specific *JavaScript functions* implementing the respective runtime support. The library of JavaScript functions can also be programmed directly by programmers, without the need for interacting with UI elements. To users, the UI elements act as proxies toward the features of the library. A UI-oriented computing *middleware* complements the library; both are part of the browser plug-in. It takes care

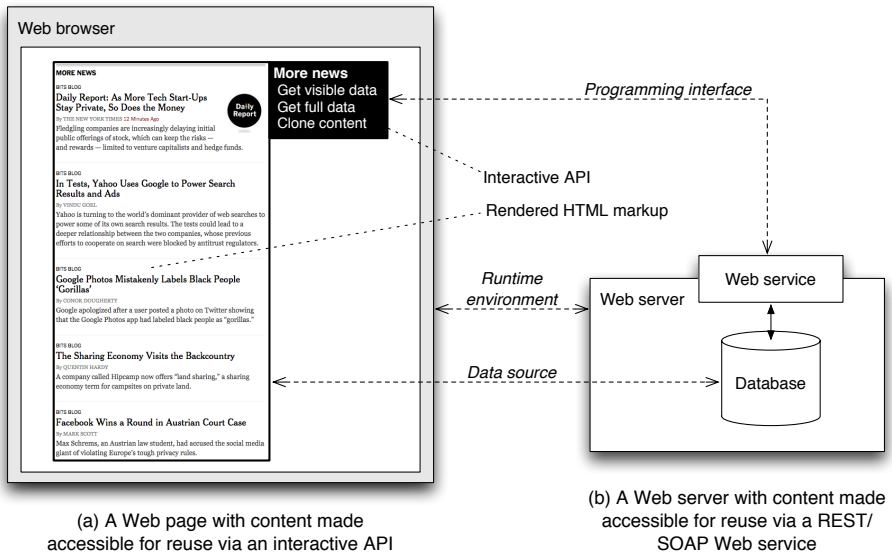

Fig. 1. Analogy between visual, interactive APIs (iAPIs) and conventional RESTful or SOAP Web services: iAPIs are executed inside the client browser and "programmed" visually and interactively via graphical controls injected into the markup of the page.

of setting up communications among integrated applications (e.g., to load data dynamically from third-party pages) and of storing interactively defined integration logics in the browser's *local storage*. Programmers with access to the source code of a page can inject their JavaScript code directly into it. If a potential source page is not yet annotated to support iAPIs, it is possible to *inject* suitable annotations from the outside and to store them either locally on a remote Web server for reuse and sharing.

For a better understanding, Figure 1 shows a possible rendering of an iAPI inside a Web page and also draws the parallelisms with conventional APIs, such as RESTful or SOAP Web services. In [8], we discuss how the graphical controls and standard user interactions like drag and drop, point and click, buttons, and similar can be interpreted as programming intentions; the paper specifically focuses on the case of data integration, the scenario we will approach in the Challenge. The paper also provides a detailed description of the iAPI annotation format used in the implementation described in this work.

3 UI-Oriented Computing Infrastructure

Figure 2 shows the internal architecture of the current prototype, which comes as a Google Chrome browser extension. It comes with two core elements: a UIC engine for the execution of UI-oriented data integration logics and an iAPI editor for visual, interactive development. The UIC engine is split into

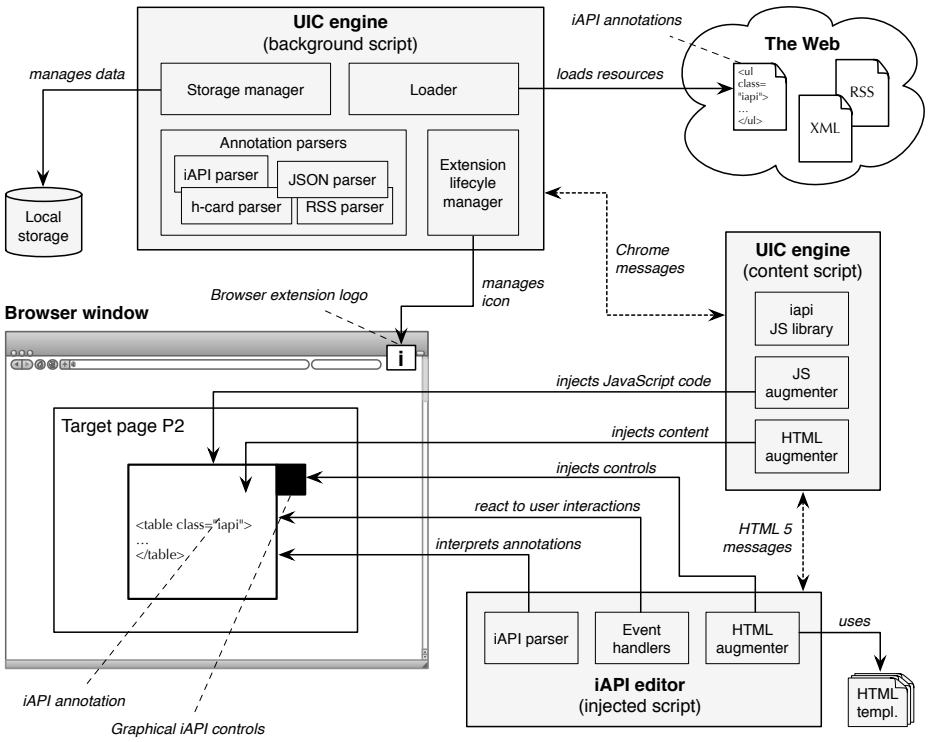

Fig. 2. Architecture of the UI-oriented computing environment as browser extension.

two parts: The *background script* provides core middleware services, such as extension management (via its icon and pop-up menu), remote resource access, data parsing, and local storage management. The *content script* implements the `iapi` JavaScript library for programmatic UIC (the implementation is based on `http://toddmotto.com/mastering-the-module-pattern`), injects JavaScript code into the page under development, and provides for the rendering of data (using the jQuery plug-in). Content and background script communicate via Chrome system messages. The iAPI editor comes as JavaScript code that is injected into the Web page under development. It parses the annotations of the iAPIs inside the page, augments them accordingly with graphical controls, and injects the event handlers necessary to intercept user interactions that can be turned into JavaScript data integration logics (in turn, injected into the page by the content script).

As for the features identified in the Call for Participation of the Rapid Mashup Challenge (`http://mashup.inf.usi.ch/challenge/2015/checklist.html`), UI-oriented computing and the current implementation of the prototypical computing infrastructure support the features summarized in Figure 3. The essence of UIC is that it aims at the development of applications without the need to code any interaction with APIs or services of the Deep Web, therefore it

specifically focuses on UI mashups. Hybrid mashups, i.e., mashups that integrate also application logic and/or data sources, are supported in that application logic can be accessed by automating and making reusable the interaction with HTML forms, and data can be extracted from Web pages (we will use both these features in the Challenge). The core component types the approach focuses on are UI components, the iAPIs, and they are integrated on the client-side inside the Web browser. Some features of the runtime environment, e.g., the persistent storage of external Web page annotations and the form automation service, are hosted on a Web server but integrated inside the client browser. The respective integration logic is UI-based, in line with the vision of UIC, and applications are short-lining. That is, they are applications running inside the client browser, and their runtime lifecyle only depends on the lifetime of the respective browser window: once closed, the application is stopped.

Regarding the features provided by the iAPI editor (the "mashup tool"), it targets end users and aims to enable them to perform simple data integration operations interactively inside their own browser. The JavaScript library for coding iAPI reuse targets programmers. The degree of automation is high for end users (programming instruction are derived automatically from their user interactions and configurations), while coding the JavaScript library is a manual effort. The liveliness level of the resulting development experience is that of dynamic modification, that is, live development inside the browser. The interaction technique proposed is WYSIWYG for the users of the iAPI editor (the results of all integration actions are rendered immediately); the recording of user interactions with forms for their automation follows a programming by demonstration approach, which is however again visual and interactive, just like the iAPI editor. Programmers, instead, can rely on a textual DSL implemented as a set of functions provided by the JavaScript library.

Checklist	
Mashup Features	**Mashup Tool Features**
Mashup Type:	Targeted End-User:
User Interface (UI) mashups	Non Programmers
Hybrid mashups	Expert Programmers
Component Types:	Automation Degree:
UI components	Semi-automation
Runtime Location:	Manual
Client-side only	Liveness Level:
Integration Logic:	Level 4 (Dynamic Modification)
UI-based integration	Interaction Technique:
Instantiation Lifecycle:	WYSIWYG
Short-living	Programming by Demonstration
	Textual DSL
	Online User Community:
	None

Fig. 3. Summary of the features by the proposed UI-oriented computing paradigm.

4 The Challenge: Scenario and Preparation

Given the set of APIs that can be used in the context of the Rapid Mashup Challenge (Google Maps, Youtube and the New York Times) and the described goals and implementation of the UI-oriented computing approach, we chose to participate in the Challenge with a data integration scenario. Next, we describe the target mashup in more details and explain how we prepared for the Challenge.

Fig. 4. The target data mashup running in the browser.

4.1 Mashup Scenario

We explain the target mashup by means of its screen shot in Figure 4. The application is a data integration that takes latest technology news from the New York Times (`http://www.nytimes.com/`) and the Discover Magazine (`http://discovermagazine.com/`) – news are represented by their title, author and summary – and also provides a translation of the summary from English to Italian using the Yandex Translation API (`https://tech.yandex.com/translate/`). The two data sources are integrated via a common merge/union operation, while the translation requires iterating over each news article and invoking the translation Web service for each summary. The result is rendered inside the target page of the developer by means of a common HTML table.

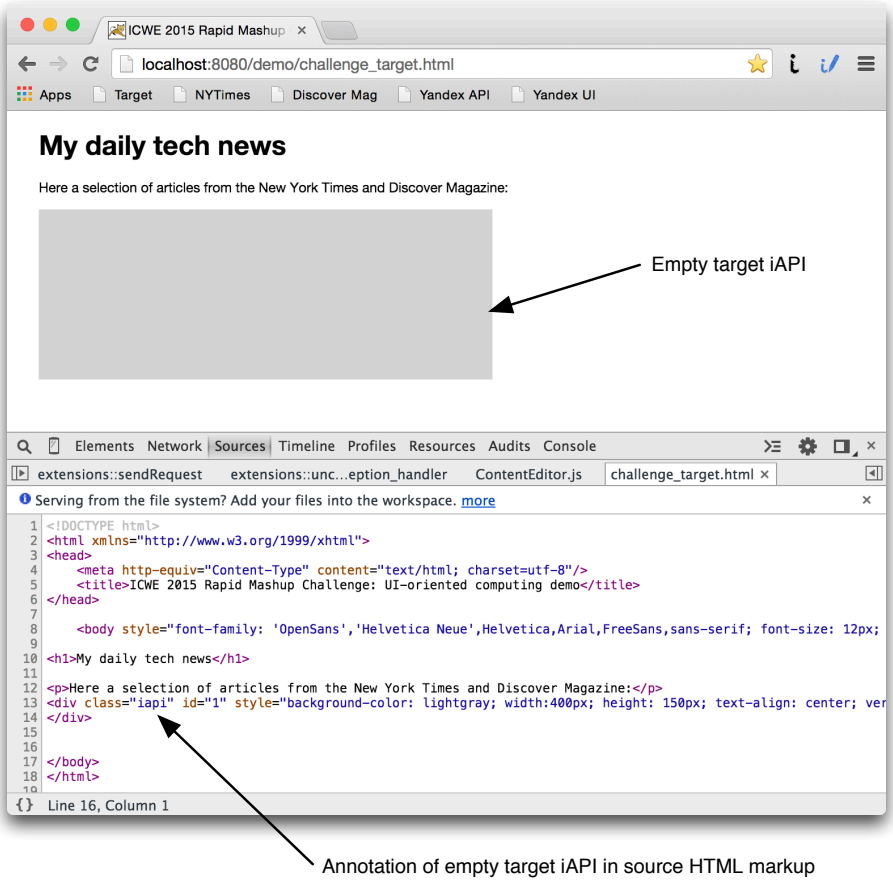

Fig. 5. The empty target mashup running in the browser.

4.2 Preparation of Challenge

Mashing up the two data sources and the translation API in the scenario with the proposed UIC paradigm requires some preparation. In general:

1. *Implementing suitable UIs for all resources.* For data and functionality to be extracted from Web pages, the UI is already there. For data feeds, services or APIs, this requires new simple Web front-ends that provide access to the resources' features, e.g., tables visualizing data from feeds or forms allowing users to operate a remote service or API.
2. *Annotating all UIs for reuse.* For existing Web pages this requires injecting annotations into the markup of the pages, e.g., using the interactive iAPI annotator (developed in parallel to the core UI-oriented computing infrastructure) that allows one to inject iAPI annotations into a page at the client-side at page loading time. Newly developed front-ends can directly be annotated in their source markup.

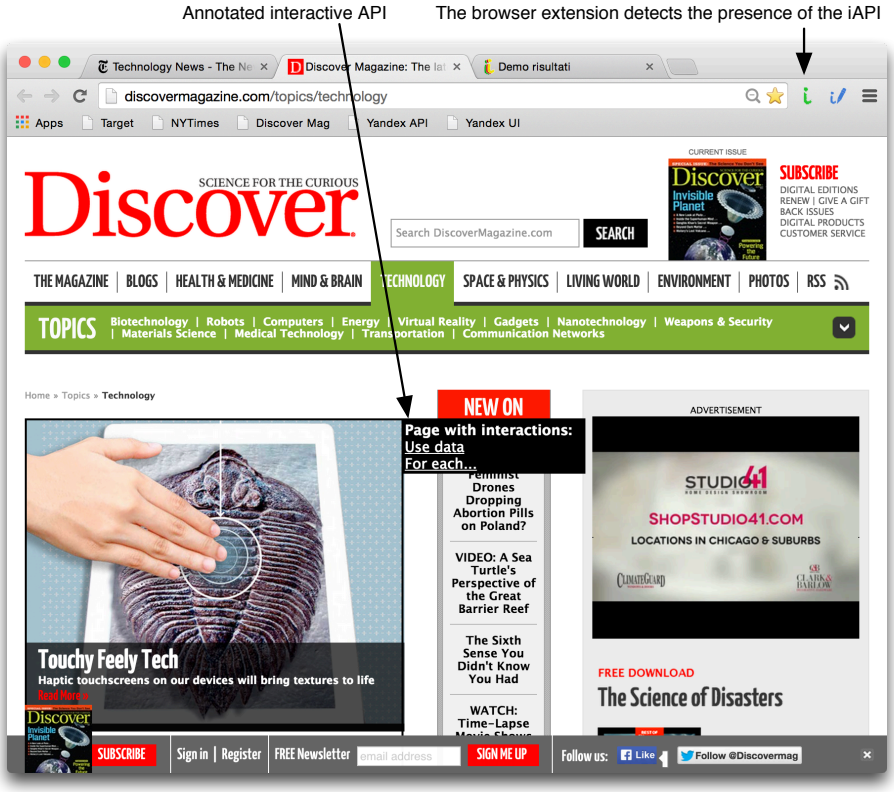

Fig. 6. The annotated Discover Magazine with injected graphical controls.

Specifically, this means that we need to annotate the Discover Magazine to enable the extraction of news and to implement an ad-hoc HTML form providing access to the translation API. In addition, we also need to implement an empty target page that will host the integrated data and translations. We do not annotate the New York Times in advance, since we also would like to demonstrate the use of the interactive iAPI annotator during the Challenge. We describe the preparation of the other parts next, starting from the target page.

The screen shot in Figure 5 illustrates the implementations of the *target page*. The top part is the rendering of the page inside the browser; the lower part reports the source HTML markup of the page. As can be seen in the code, the page does not have any own data to be rendered, and the gray shaded `div` element is marked as an interactive API by the annotation `class="h-iapi"`. This simple annotation is enough to turn the `div` into a UI element users can interact with. In our case, this is the UI element that will host the integrated data. Nothing more is needed to implement the target page.

Figure 6, instead, illustrates the annotated start page of the *Discover Magazine*. The annotation is achieved by means of the iAPI annotator tool, which

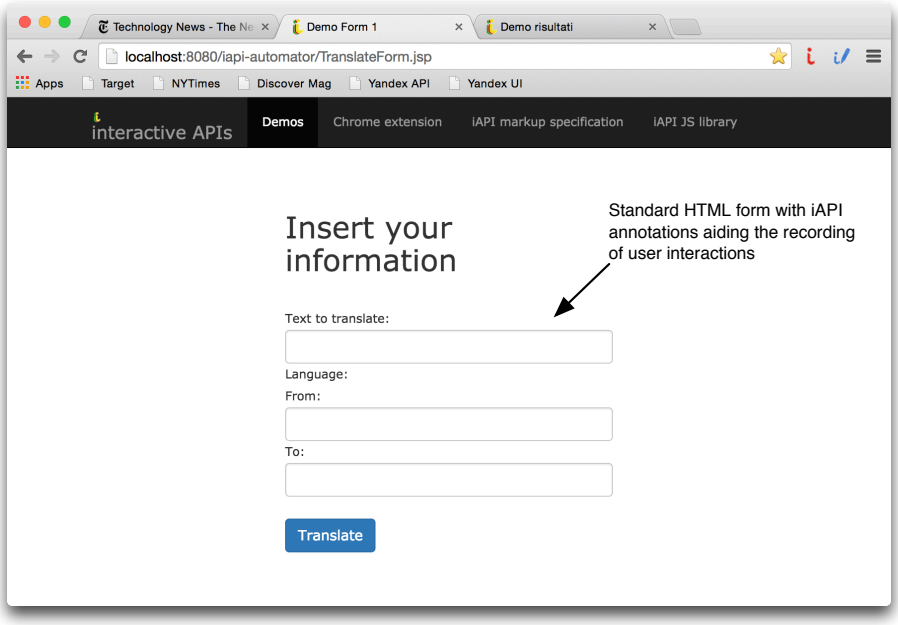

Fig. 7. The auxiliary HTML form developed on top of the Yandex translation API to enable UI-oriented reuse.

allows one to annotate interactively a page and to inject annotations on the fly each time the annotated page is accessed. This means that the annotation of the magazine does not require us to download the page and to store it locally; instead only the annotations are stored in a dedicated Web-accessible repository and reused at each access to the page. The specific annotations used to extract news from this page are (used in the `class` attribute of HTML elements):

- `h-iapi`: identifies the area from which to extract content;
- `e-data:News`: categorizes the identified iAPI as a data source and labels it as "News;"
- `e-item:Article`: identifies the DOM nodes that host individual news items and assigns the label "Article" to them;
- `p-attr:Title`, `p-attr:Author`, `p-attr:Summary`: identify the different components that make up a news item (the attributes of the item) and labels them as "Title", "Author" and "Summary."

The same annotation structure will be used during the Challenge to annotated the New York Times news items. This allows us to automatically match items at data integration time without the need for transforming input data structures and to save time during the live demonstration.

Finally, Figure 7 shows the HTML form developed on top of the *Yandex Translation API* (a RESTful Web service). Since we do not directly want to

interact with the API itself, the form is needed to make its functionality available through the Surface Web. The form comes with three input fields (text to translate and the input/output languages) that allow the user to translate text by invoking the translation API in the background on behalf of the user. The result is shown on another page after hitting the Translate button. In the next Section, we will see how this form can be programmed by example and turned into a piece of reusable business logic for the development of the target mashup.

5 The Challenge: Live Mashup Development

Given the empty target page, the annotated Discover Magazine and the HTML form that provides interactive access to the translation API, we are ready for the development of the mashup to be showcased in the Challenge. The available time to showcase the UI-oriented computing approach and to develop the mashup outlined above is 10 minutes. We structure the demo into the following steps:

1. Annotation of the New York Times technology news
2. Fetching of news from the New York Times
3. Fetching of and merging with news from the Discover Magazine
4. Rendering of integrated data suing a table representation
5. Programmatic addition of a new column to host the translations
6. Recording of user interactions with the translation form for reuse
7. Programmatic iteration over news and reuse of recorded interactions
8. Rendering of integrated dataset

Next, we describe the demo showcased during the Mashup Challenge step by step and provide the necessary explanations with the help of screen shots.

Figure 8 illustrates the annotation process for the New York Times technology news (❶). We specifically focus on the "More news" area, which is well structured and allows us to easily annotate and extract news items. Clicking on the "i" icon with the pencil in the top right corner of the browser opens the overlay window shown in the lower right part of the screen shot. This window serves as control console for the annotation process. The process is as follows: First, the user identifies the HTML area of interest (this is highlighted in the left-hand side of the screen shot by the rectangular box surrounding the news to be extracted). Then, the user identifies the DOM element that hosts an individual news article (represented by the green-shaded area in the top part of the highlighted area inside the page). The annotator tool automatically identifies all DOM elements with similar structure. Next, the user identifies the individual attributes of each news item by selecting them inside one of the identified news items. Once all attributes are identified, the control panel allows the user to label the data source ("News"), the items ("Article") and the attributes ("Title", "Author", "Summary"). Finalizing the annotation process saves the annotations using a dedicated Web service and injects them into the page. The newly created iAPI is ready for data extraction.

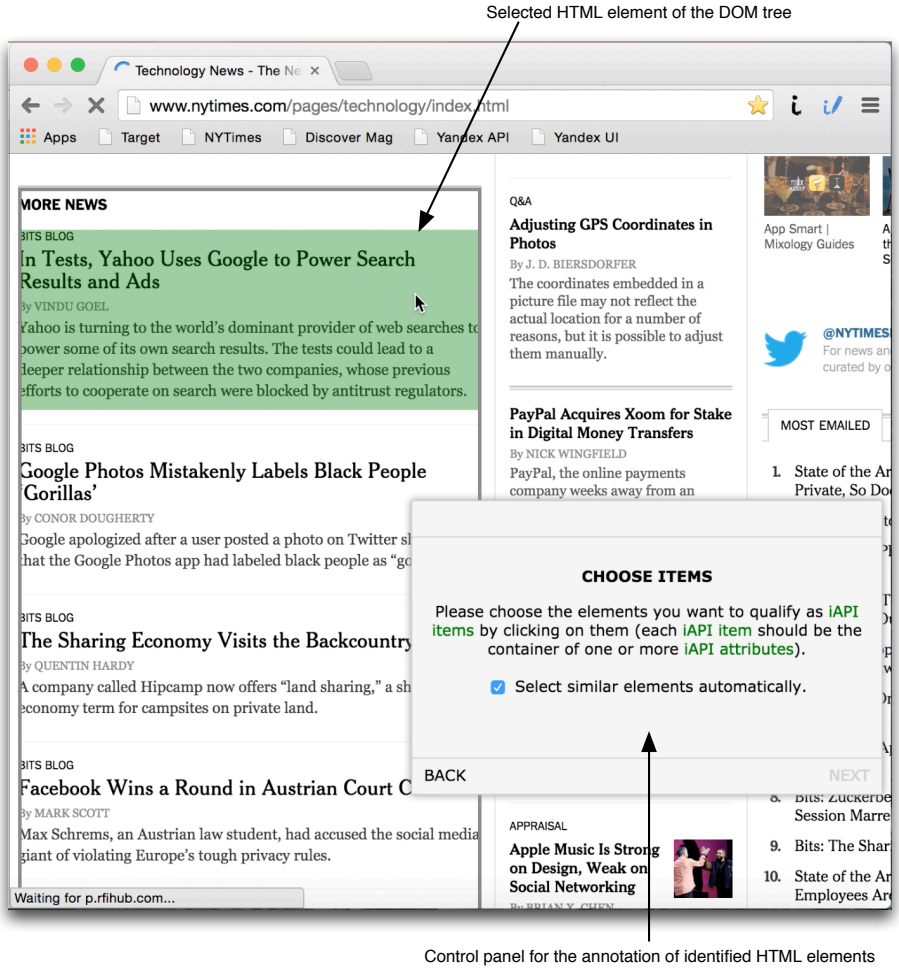

Fig. 8. Interactive annotation of the New York Times Technology News site.

The reuse of the identified news articles (❷) is now supported via a simple drag and drop action. Figure 9 illustrates the process. When the user moves the mouse over the area marked as iAPI inside the New York Times page, the black graphical controls pop up and allow him/her to pick the data by dragging and dropping the "Get data" menu entry of from the injected menu. Since the target iAPI is still empty, this process fills the iAPI with the extracted data.

The next step of the data integration process (❸) requires the user to repeat a similar drag and drop action using the Discover Magazine, as illustrated in Figure 10. The key difference from the first action is that now at drag release time the target iAPI allows the user to specify how to disambiguate his/her action (in fact, multiple interpretations of a drop action on an iAPI that already contains data are possible, e.g., join, merge, substitute, etc.). In our scenario,

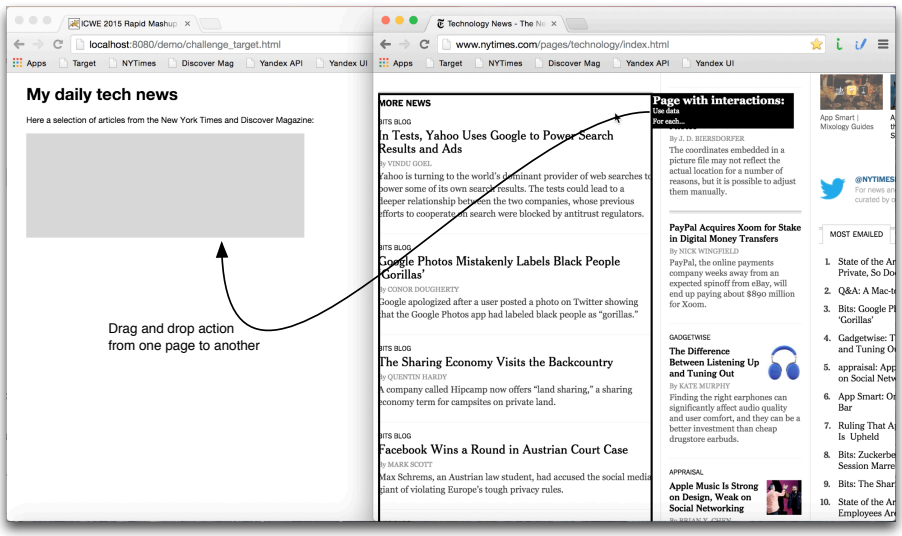

Fig. 9. Dragging and dropping news articles from the New York Times into the target page fills the target iAPI with extracted data and applies a standard visualization format, e.g., a list or table layout.

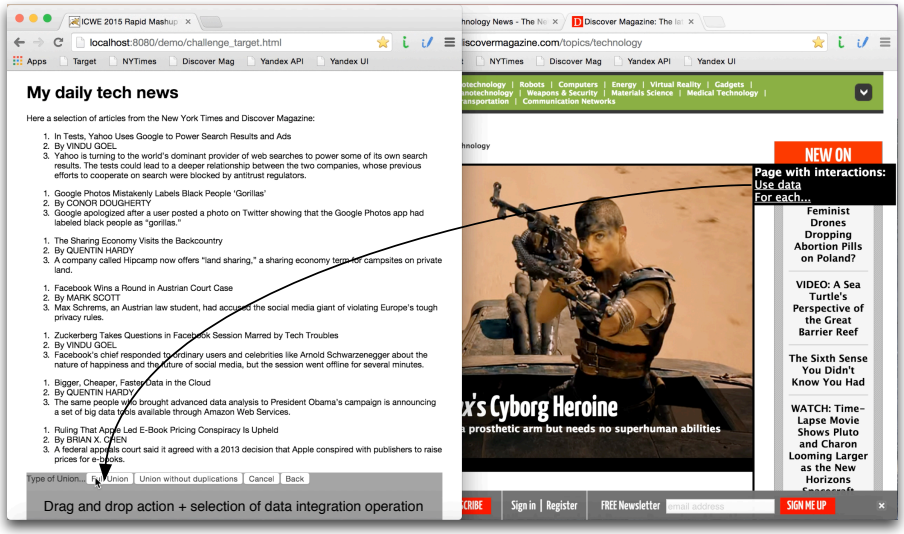

Fig. 10. Dragging and dropping news articles from the Discover Magazine into the target page causes the target iAPI to ask the user which action he/she wants to perform, given that there are already data in the iAPI.

the user chooses to "merge" the new data with the one already fetched from the New York Times, specifically using a "full union" operator (there is no need to eliminate possible duplicates, as the two data sources are too different and it is unlikely that there will be two articles with exactly the same title, author and summary). A final selection of the table layout from the injected menu of the target iAPI reformats the data fetched from the two data sources as illustrated in the top part of Figure 11 (❹).

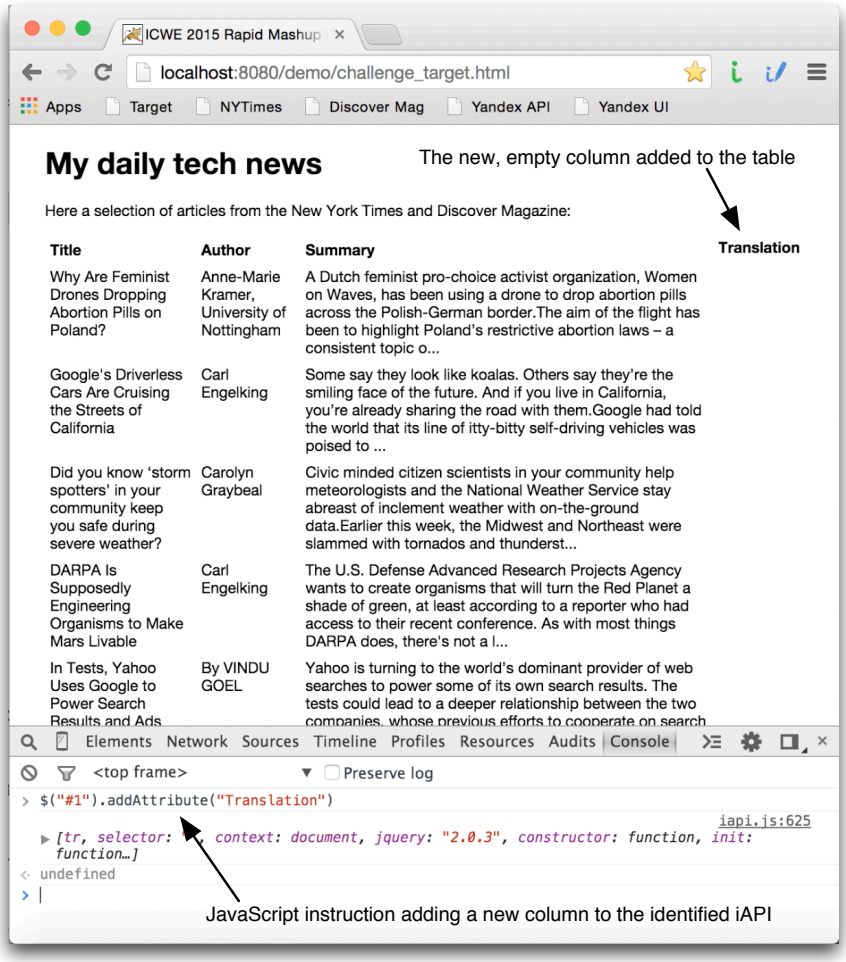

Fig. 11. Programmatic extension of the table with a new column for the translation

To showcase how programmers can leverage on the proposed UI-oriented computing paradigm, we now switch off the interactive iAPI editor that injects graphical controls using the pop up menu that opens when clicking on the ex-

tensions logo in the top right of the browser window and turn on the JavaScript console of the browser. This allows the skilled programmer to input UI-oriented programming instructions in JavaScript and to modify the mashup rendered in the browser window on the fly.

The screen shot in Figure 11 illustrates the first step of the manual development process, i.e., the expansion of the table in the browser with a new column able to host the translations of the summaries (❺). The JavaScript console reports the respective programming instruction. The selector `$("#1")` is the jQuery (`https://jquery.com/`) selector that uniquely identifies the target iAPI inside the target page (see Figure 5). The function `addAttribute` injects the new column into the iAPI, both into its in-memory data object and its graphical rendering inside the page.

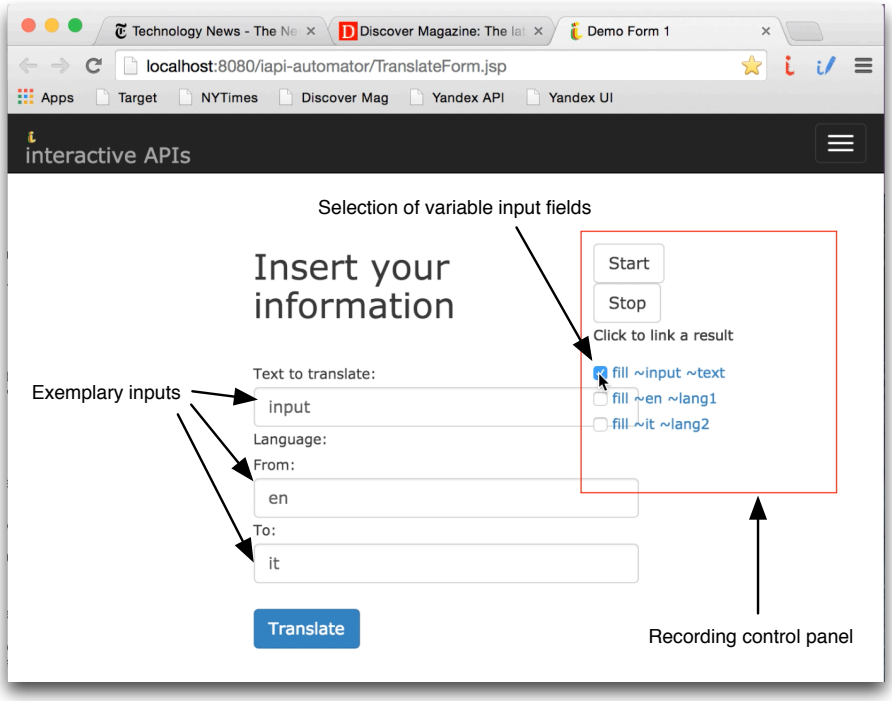

Fig. 12. Recording user interactions with the HTML form providing access to the Yandex translation service. The controls at the right allow the user to start/stop the recording and to identify variable inputs to be filled at invocation time.

The next step is the translation of the summaries. Doing so requires first recording an exemplary interaction with the translation HTML form we prepared before the Challenge (❻). This process is illustrated in Figure 12. The recording control panel allows the user to start and stop the recording and to mark input

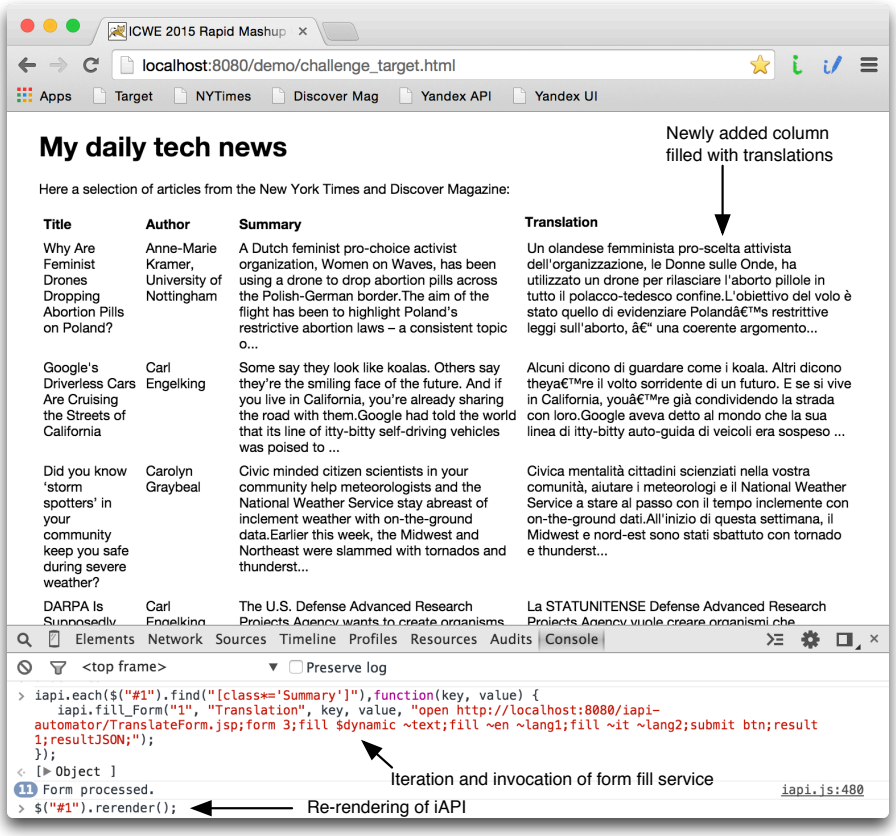

Fig. 13. Iteration over all articles and invocation of the translation form for each summary with final re-rendering of the target iAPI.

fields as either constants (the values provided as examples during the recording will also be used when replaying the recorded interactions) or variables (the values of these can be provided as dynamic inputs each time recorded interactions are replayed). A click on the Translate button invokes the Yandex translation service and renders the translated text. This latter can now be indicated as output of the recorded interaction process. A click on the Stop button terminates the recording and opens a pop-up window that provides the user with a simple script that can be used to invoke the recorded user interactions. This script is shown in Figure 13 in the JavaScript console (the string in red) and used inside an `iapi.each` iterator that scans all news articles in the table and allows the invocation of the `iapi.fill_Form` function that mimics the filling of the translation form for each summary found in the table (❼). The final re-render instruction in the JavaScript console renders the retrieved translations (❽), and closing the console brings us to the final mashup already shown in Figure 4.

The eight described steps showcase how the UI-oriented computing paradigm has been implemented so far for both users and programmers. The video available at `http://youtu.be/yEtjIO3oMsI` shows the screen cast of the demonstration and provides better insight into the subjective experience of both types of developers.

6 Related Work

The key idea of UI-oriented computing is to interpret standard UI elements – like the ones already in use for the implementation of Web UIs – as constructs to express generic computation logics. Traditionally, computation logic for the Web is expressed either via programming languages, such as Java, Python, PHP, JavaScript, and similar, or via model-driven development formalisms [6]. Orthogonally to these paradigms, Web services [2, 12] have emerged over the last decade as one of the most prominent Web technologies that influenced integration on the Web in general. Their focus, however, is on the application logic layer, not the presentation layer (the UIs) of applications.

Research on the reuse of UIs has mostly focused on the identification and definition of UI-centric component technologies, such as standard W3C widgets [14] and Java portlets [13] or proprietary formats [15], and the development of suitable integration environments [5, 7]. The former essentially apply the traditional programmer perspective to UIs and still require integration at the application logic layer, e.g., via Java or JavaScript. The latter generally follow a black-box approach in the reuse of UIs: components are small, stand-alone applications and they are either included or excluded in a composition/workspace. The Web augmentation approach by Diaz et al. [11] is a partial exception: it allows for a fine-grained reuse of data among websites, starting from their UIs. The approach extracts data elements of limited size (individual labels or small fragments) without requiring additional annotations; on the downside, the approach still requires programming knowledge. None of these UI-centric approaches are however able to implement the data integration scenario approached in this paper.

Mashups [10] are the approach that comes closest to the described scenario; in fact, the discussed data integration can be seen as a mashup, in particular, a data mashup. It could, for instance, be approached with the help of Yahoo! Pipes, JackBe Presto, or similar data mashup tools. Pipes (`http://pipes.yahoo.com`), for example, proposes a model-driven paradigm that starts from the assumption that the data to be integrated are available as RSS/Atom feeds or XML/JSON resources. The two lists of news articles integrated in our example scenario could thus be merged by selecting and configuring dedicated built-in constructs; the translation of the summaries would however require some manual development of a back-end Web services compatible with Yahoo! Pipes data passing logic (in complete lists, not individual items). The result would then be accessible as RSS feed via Yahoo! Pipes. Although the described logic is very similar to the one of our scenario, it still lacks the rendering and embedding of the result into the user's website, a task that requires again considerable manual development.

To aid both the extraction of content from HTML markup and the transparent invocation of backend Web services, this paper proposes the use of explicit annotations, similar to microformats (`http://microformats.org`). If these are not provided natively inside of the markup of a source page (as in the case of the form we annotated for the reuse of the RESTful translation service), the iAPI Annotator provides the necessary means to attach them from the outside to third-party pages (as in the case of the New York Times). The approach does not yet focus on the annotation of data with semantics, as proposed by the Semantic Web initiative [3]. The goal of the annotations in this work is to provide immediate functional benefits to the consumers of data: annotations in fact allow the injection of graphical controls that enable the visual UIC paradigm.

7 Discussion and Future Work

The demo showcased in the context of the Rapid Mashup Challenge and described in this paper is the development of a simple data mashup following a UI-oriented computing approach. The idea of the approach is to leverage on the graphical UIs of applications as programming artifacts, to extend them with additional, programming-specific controls, and to allow developers (both common users and programmers) to express data integration operations interactively inside the browser without having to write any line of program code. The idea of UI-oriented computing and interactive APIs is still in its infancy. Yet, the demo – although apparently simple – showed a data integration scenario that is not trivial in general but that was solved in a fashion that does not require programming skills (the first part of the demo) or manually programming low-level interactions with Web services or data extractors (second part of the demo). The benefits of the approach therefore span from common users to skilled programmers.

There are however still some limitations that come with the showcased implementation of the UI-oriented computing infrastructure and the iAPI editor:

- The current implementation of the editor does not yet support the visual specification of iterators and the reuse of recorded user interactions for the automation of forms. We turned this shortcoming in the demo into an advantage and used it to also showcase how programmers can leverage on the proposed paradigm. This was possible thanks to the ready implementation of the respective functionality in the `iapi` JavaScript library. The next step is however making the these features available also to regular users through the interactive iAPI editor.
- The interaction paradigm proposed in this paper and the demonstration to derive programming intentions from user interactions is a best-effort development. We did not yet have time to study different types of interpretations (e.g., whether a drag and drop action better represents a data fetching action or a layout action) or different interaction paradigms (e.g., without drag and drop actions, with contextual menus that can be opened with a right-click, voice interactions, etc.). However, the current implementation of the

described software infrastructure already supports the independent development of different editors on top of the runtime environment, which will ease these kinds of investigations in future developments.
- The annotation format proposed so far to equip UIs with interactive programming capabilities, the interactive APIs, does not leverage on any form of semantic knowledge. The format is inspired by the microformats 2 proposal (`http://microformats.org/`) and provides syntactic cues for the runtime environment only. We are aware that especially targeting end users without specific programming skills may require better assistance mechanism, able to provide them with as much aid as possible. Doing so may require using also semantic annotations, e.g., in order to automate some data integration tasks (most notably, data disambiguations).
- The UI-oriented computing features supported so far are mostly focused on data integration tasks, with the exception of the user interaction recorder that allows interpreting standard HTML forms as reusable pieces of business logic. The idea of UI-oriented computing is however much broader and comprises also use cases for cloning complete UI widgets (markup, styles and functionality), automating short-living and long-living processes (e.g., the parametric execution of repeated navigation actions), and the establishment of communications among integrated widgets or UI elements. These advanced use cases are part of our future work.

As these considerations point out, UI-oriented computing is not a pure engineering problem only. Identifying the right set of operations and use cases that make sense in a UI-only context, understanding how to best interpret user intentions, designing effective interaction paradigms, etc. are all HCI challenges that need good answers on their own. Of course, the engineering of the necessary software support inside and outside of the browser requires profound software engineering and Web development skills. The challenge of the proposed idea is finding the right answers in both areas and to bring them together profitably. The final vision of iAPIs and UI-oriented computing is proposing an alternative to the current interpretation that programming is only for skilled programmers that can only be achieved by means of abstractions and constructs that only programmers are familiar with and can master. That is, the vision is to make "programming" accessible to an increasingly wider area of "developers."

What makes us confident about the potential success of UI-oriented computing is that, although it's final vision targets non-programmers, it also immediately provides tangible benefits the programmers: The deployment of iAPIs is contextual to the deployment of their host application, and they do not require separate deployment or maintenance (like, for instance, the RSS feeds published by the New York Times in parallel to the main Web site). The documentation of iAPIs comes for free; the UI and the injected graphical controls already tell everything about them. The retrieval of iAPIs does not ask for new infrastructure or query paradigms; since iAPIs are an integral part of the Surface Web, it is enough to query for desired data or functionality via common Web search; if Google indexes a Web site, its iAPIs are indexed too.

The iAPI microformat is maintained via the W3C Interactive APIs Community Group (`http://www.w3.org/community/interative-apis`), the browser extension on `https://github.com/floriandanielit/interactive-apis`.

References

1. A. Abdelnur and S. Hepper. Java Portlet Specification, Version 1.0. Technical Report JSR 168, Sun Microsystems, Inc., `http://download.oracle.com/otndocs/jcp/PORTLET_1.0-FR-SPEC-G-F/`, October 2003.
2. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures, and Applications*. Springer, 2003.
3. T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, pages 34–43, May 2001.
4. M. Caceres. Packaged web apps (widgets) - packaging and xml configuration (second edition). *W3C Recommendation*, 2012.
5. C. Cappiello, M. Matera, M. Picozzi, G. Sprega, D. Barbagallo, and C. Francalanci. DashMash: A Mashup Environment for End User Development. In *ICWE 2011*, pages 152–166, 2011.
6. S. Ceri, P. Fraternali, A. Bongio, M. Brambilla, S. Comai, and M. Matera. *Designing Data-Intensive Web Applications*. Morgan Kauffmann, 2002.
7. O. Chudnovskyy, T. Nestler, M. Gaedke, F. Daniel, J. I. Fernández-Villamor, V. I. Chepegin, J. A. Fornas, S. Wilson, C. Kögler, and H. Chang. End-user-oriented telco mashups: the OMELETTE approach. In *WWW 2012 (Companion Volume)*, pages 235–238, 2012.
8. F. Daniel. Live, Personal Data Integration through UI-Oriented Computing. In *ICWE 2015*, pages 479–497, 2015.
9. F. Daniel and A. Furlan. The Interactive API (iAPI). In *ComposableWeb 2013 (ICWE 2013 Workshops)*, pages 3–15. Springer, July 2013.
10. F. Daniel and M. Matera. *Mashups: Concepts, Models and Architectures*. Springer, 2014.
11. O. Díaz, C. Arellano, and M. Azanza. A Language for End-user Web Augmentation: Caring for Producers and Consumers Alike. *ACM Trans. Web*, 7(2):9:1–9:51, May 2013.
12. R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Ph.d. dissertation, University of California, Irvine, 2007.
13. S. Hepper. Java Portlet Specification, Version 2.0, Early Draft. Technical Report JSR 286, IBM Corp., `http://download.oracle.com/otndocs/jcp/portlet-2.0-edr-oth-JSpec/`, July 2006.
14. Web Application Working Group. Widgets Family of Specifications. Technical report, W3C, `http://www.w3.org/2008/webapps/wiki/WidgetSpecs`, May 2012.
15. J. Yu, B. Benatallah, R. Saint-Paul, F. Casati, F. Daniel, and M. Matera. A Framework for Rapid Integration of Presentation Components. In *WWW 2007*, pages 923–932, 2007.