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 live 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 and Goals

The most notable technologies today to publish and access data and functionality over the Web are SOAP/WSDL Web services [2], RESTful Web services [6], 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 [3] 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 [4]) 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 [5], 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 UIs. In [4], 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 to be 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).

2 UI-Oriented Computing Approach

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 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.

3 UI-Oriented Computing Infrastructure

Figure 1 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 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).

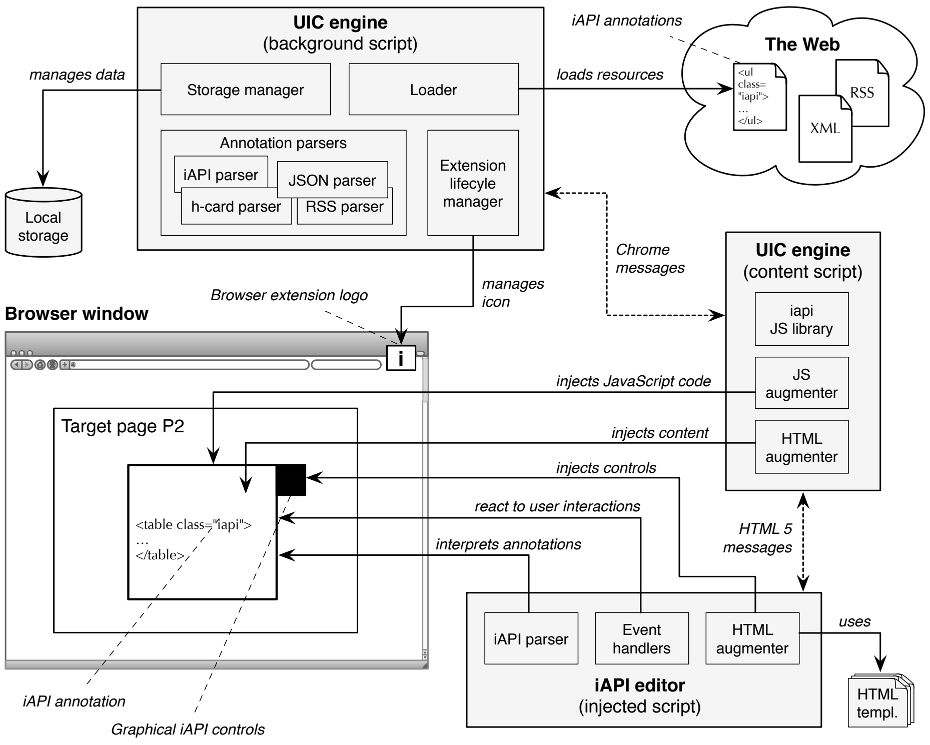

Fig. 1. Architecture of the UI-oriented computing environment as browser extension.

4 Preparation and Demonstration

Once the resources to be integrated in the context of the Challenge are available, mashing them up with the proposed UIC paradigm requires three steps:

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. Newly developed front-ends can directly be annotated in their source markup.
3. Integrating them inside the Web browser.

The first two points will be done as part of the preparation of the Challenge; the latter will be showcased live during the challenge, first interactively inside the Web browser (e.g., `https://www.youtube.com/watch?v=9CRKzToL7tc`), then programmatically using the JS library oriented toward programmers (e.g., `https://www.youtube.com/watch?v=h3C-YEMUxG0`). The challenge will be to make available through the Surface Web as many resources as possible and to understand which mashup scenarios suit the envisioned UIC paradigm best.

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`.

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

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. M. Caceres. Packaged web apps (widgets) - packaging and xml configuration (second edition). *W3C Recommendation*, 2012.
4. F. Daniel. Live, Personal Data Integration through UI-Oriented Computing. In *ICWE*, 2015.
5. F. Daniel and A. Furlan. The Interactive API (iAPI). In *ComposableWeb 2013 (ICWE 2013 Workshops)*, pages 3–15. Springer, July 2013.
6. R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* Ph.d. dissertation, University of California, Irvine, 2007.