The Interactive API (iAPI)

Florian Daniel and Andrea Furlan

University of Trento, Via Sommarive 5, 38123, Trento, Italy
daniel@disi.unitn.it, andrea.furlan@studenti.unitn.it

Abstract. The claim of this paper is that *reuse on the Web* – if sensibly facilitated – can be achieved in a much more intuitive and efficient fashion than today. The idea is to invert the current perspective on reuse by moving away from programmer-oriented artifacts, such as APIs, web services and data feeds, and focusing on user-oriented artifacts, i.e., graphical user interfaces (UIs). The paper defines a new kind of API, the *interactive API (iAPI)*, which reconciles the intuitiveness of interactive UIs with the power of programmable APIs and enables (i) *programmatic access* to UIs and (ii) *interactive, live programming.* The paper discusses use cases and implementation options and lays the foundation for *UI-oriented computing* as a discipline.

1 Introduction

On the Web, *data* is typically shared via XML dialects like RSS [9] or Atom [8] feeds, plain XML, web services or, more recently, via so-called micro-formats (`http://microformats.org`), which enable the extraction of structured data from web pages via HTML annotations. If data are not explicitly made accessible, data extraction tools like Dapper (`http://open.dapper.net`) allow one to extract structured data from websites even without annotations. *Application logic* is mostly accessed via web services (both SOAP services [2] and RESTful services [4]) or code libraries (e.g., in JavaScript). The reuse of *user interfaces* (UIs) is only scarcely supported so far. Web mashups [11] are the most prominent example of integration at the presentation layer. W3C widgets [10] are a standard client-side technology, Java portlets [1] a standard server-side UI technology for simple, stand-alone applications that can be assembled into a composite page. However, mashup tools typically still rely on proprietary UI component technologies [12] and, like portlets and widgets, feature only the reuse of coarse-grained, programmer-oriented components.

The promise of the service composition (e.g., BPEL [5]) and mashup approaches [11] was that they would enable generic people to develop. Yet, if we take a critical look at how composite applications and services are developed today we recognize that development is still a prerogative of programmers. Mastering all the above technologies implies a learning process that is tedious and time-consuming even to ***expert programmers***, let alone ***non-programmers***. All the attempts to enable non-programmers failed, as they insisted on abstracting APIs or services that were invented for programmers. Non-programmers simply don't know what services or data formats are [7]. What they know is how to use a user interface.

The observation of this paper is that there is however a variety of applications on the Web, whose development could be significantly *sped up* to programmers and *enabled* to non-programmers, if only we had the right component technology in place. The ***types of applications*** we have in mind are non-mission-critical applications, such as:

– *Simple application integrations*, which require interoperability among web applications;
– *UI-centric web mashups*, which require sourcing and combing data, application logic and/or UIs from the Web;
– *Website evolutions*, which require restructuring or extending existing websites;
– *Simple web automations*, which require batch processing of repetitive on-line tasks (beyond `http://ifttt.com`); and
– *Personal web processes*, which require support for long-lasting on-line tasks that also involve user interactions.

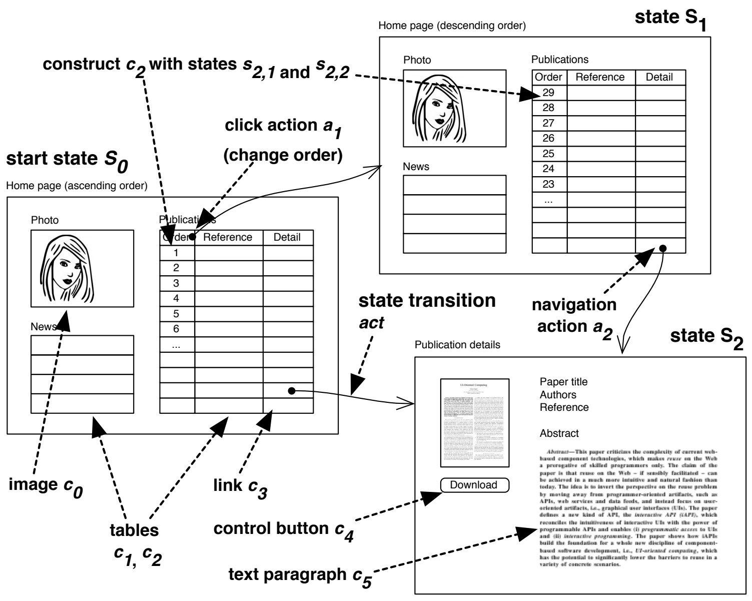
The practices underlying these applications are sourcing or extracting *data* from websites, processing data and accessing remote *application logic*, copying and pasting pieces of *UIs* from existing websites, and similar – all tasks that are relatively common on the Web. None of these practices, however, is currently supported by a *single* component technology and in the reach of *non-programmers.*

A simple, but good example of how reuse *could* be is the following ***scenario***: Imagine a researcher wants to re-structure her website, reusing the list of publications of her old website and adding citation counts from Google Scholar. She would like to focus only on the design of the new layout of her publications and to be able to simply drag and drop the content of her old table of publications to the new layout. Next, she would like to be able to tell her new website to query Google Scholar for each of the publications in the table, e.g., by recording a set of actions that she exemplarily performs manually and that her website could replay for each publication – ideally, everything without writing any new line of code.

As the scenario shows, the aim of this work is to turn UIs into first-class programming artifacts. To this aim, the paper specifically provides the following ***contributions***:

– The design of a new type of API, the *interactive API (iAPI)*, which enables (i) *programmatic access* to UIs and (ii) *interactive programming*;
– The discussion of the core iAPI *uses cases* and *implementation options*;
– The proposal of a simple *iAPI annotation format* and *runtime middleware*;
– The proposal of *UI-oriented computing* as a discipline of component-based development for non-programmers based on iAPIs.

Next, we define the context of this work, i.e., web user interfaces. Then, we introduce iAPIs and describe their use cases and a possible implementation. We then outline the benefits of iAPIs and the challenges of UI-oriented computing.



Abstract—This paper criticizes the complexity of current web-based component technologies, which makes *reuse* on the Web a prerogative of skilled programmers only. The claim of the paper is that reuse on the Web – if sensibly facilitated – can be achieved in a much more intuitive and natural fashion than today. The idea is to invert the perspective on the reuse problem by moving away from programmer-oriented artifacts, such as APIs, web services and data feeds, and instead focus on user-oriented artifacts, i.e., graphical user interfaces (UIs). The paper defines a new kind of API, the *interactive API (iAPI)*, which reconciles the intuitiveness of interactive UIs with the power of programmable APIs and enables (i) *programmatic access* to UIs and (ii) *interactive programming*. The paper shows how iAPIs build the foundation for a whole new discipline of component-based software development, i.e., *UI-oriented computing*, which has the potential to significantly lower the barriers to reuse in a variety of concrete scenarios.

Fig. 1. UI constructs, states, actions and state transitions in web UIs

2 Preliminaries: Web User Interfaces

Figure 1 provides a schematic example of a *graphical web user interface* made of two pages (Home and Publication details) and a set of UI constructs (tables, images, buttons, links, text, etc.). The figure apparently shows three pages, but attention: the Home page is shown twice. From a UI point of view it can be in two different states, depending on the order of the publications (ascending vs. descending). *State* is an important aspect of UIs, as in each instant of time a user can perform only those actions that are supported by the UI constructs available at that time (e.g., if the publications are in ascending order, he can only ask for the descending order).

Adapting traditional human-computer interaction and system models [3] to the Web, a graphical web ***user interface*** can be seen conceptually as an extended finite state machine of the form $ui = \langle C, S, S_0, A, act\rangle$, where:

- $C = \{c_i\}$ is the set of all *UI constructs* of the UI;
- $S = \{S_j | S_j = \{s_{i,j}\}\}$ is the set of *states* the UI may traverse in response to user interactions, with $s_{i,j}$ being the state of the i-th UI construct in state j (e.g., hidden);
- S_0 is the *start state* of the UI;
- $A = \{click, type, drag, drop, ...\}$ is the set of typical *actions* a user can perform on web UIs; and
- $act : C \times S \times A \to S$ is the *transition function* that tells how the UI transitions from one state to another in response to an action $a \in A$ on a construct $c \in C$.

The peculiarity of this model is that states are not atomic but *aggregations* of states of UI constructs, i.e., $S_j = \{s_{i,j}\}$. That is, a state change may correspond to the contextual change of the state of multiple UI constructs. For instance, a user navigation typically affects multiple UI constructs contemporarily, e.g., it may create a new table of data, load new images, display new heading and text, expose new interactive controls, etc.

The model does not prescribe any specific *level of granularity* regarding the set of constructs C, which means that a given $c \in C$ can be a full table, just as it can be an individual cell of the table or a sub-element of the text inside the cell. The best level of detail is the one that captures those aspects of the dynamics of a UI that are of interest, and nothing more.

This model of UIs is *conceptual* and aims to understand how to turn UIs into programming artifacts. In practice, Web UIs are rendered out of HTML markup, UI state is managed by the browser via the DOM (Document Object Model), and state transitions correspond to changes to the DOM, e.g., due to programmatic modifications or user navigations (yielding a new DOM) – everything managed by the web browser "for free." The aim of the model is to provide an interpretation of this low-level HTML/DOM model oriented toward the users of the UI, to equip it with user-oriented semantics, and to understand what it actually means to enable users to manipulate UIs instead of syntax elements.

3 The Interactive API (iAPI)

The problem with web UIs is that they do not have enough machine-processable *meaning*, which could be used to enable their interactive manipulation and programming. HTML constructs like `<table>` or `<ul>` are only syntactical markup. Constructs of more advanced UI markup languages, such as XUL (`https://developer.mozilla.org/en/docs/XUL`) or XAML (`http://msdn.microsoft.com/en-us/library/ms752059.aspx`), only format and arrange UI elements.

There are two key ingredients that UIs currently miss: once rendered, they cannot be *programmed* in a principled fashion (of course, it is always possible to hack into the UI markup and inject JavaScript code or extract data, but this is neither good practice nor does it produce good results in general) and it is not possible to use them as *design constructs* while already rendered in the browser.

What is needed is graphically illustrated in Figure 2, which focuses on the table of publications of Figure 1: the table's UI must be complemented with a dedicated *API*, which provides programmatic access to the table, and with a set of *graphical controls*, which allow the user to operate the API interactively and, thereby, to program the table. Depending on the purpose of the specific UI construct or set thereof, the API's capability can be more or less complex. For instance, the API in Figure 2 allows one to emulate user actions on the table (`do`), to extract the visible data, to source the full data underlying the table or to clone the table along with its data. In the case of a form, the API will provide for the *processing of data* in input and produce results in output; the API may

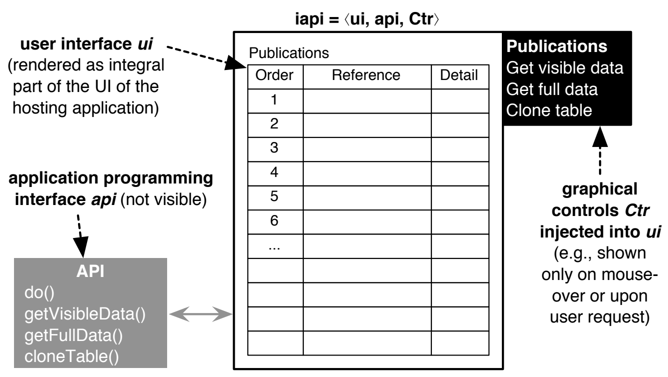

iapi = ⟨ui, api, Ctr⟩
user interface ui (rendered as integral part of the UI of the hosting application)
application programming interface api (not visible)
API
do()
getVisibleData()
getFullData()
cloneTable()
Publications
Order
Reference
Detail
1
2
3
4
5
6
...
Publications
Get visible data
Get full data
Clone table
graphical controls Ctr injected into ui (e.g., shown only on mouse-over or upon user request)

Fig. 2. The model of interactive APIs exemplified with a possible rendering

also communicate user interactions via suitable *events*, and so on. We call the artifact that brings these aspects under one hood an *interactive API.*

3.1 iAPI Model

An ***interactive API (iAPI)*** is a piece of graphical user interface (a sub-tree of the DOM), e.g., a table, form or sub-area of a web page, that provides both *interactive* and *programmatic* access to its UI constructs, application logic and/or data. That is, to the common user an iAPI is a visual What-You-See-Is-What-You-Get (WYSIWYG) artifact that can be manipulated via dedicated, artifact-specific graphical controls (see Figure 2), which may also mediate between the user and possible back-end logics (e.g., exposed via web services). To the developer, an iAPI in addition also exposes a programmable API, e.g., in JavaScript, which enables programmatic access.

Conceptually, an iAPI can be modeled as a tuple $iapi = \langle ui, api, Ctr\rangle$, where:

– $ui = \langle C, S, S_0, A, act\rangle$ is a *UI* as defined previously, but limited to the UI constructs of interest to the iAPI;
– $api = \langle do, O, E\rangle$ is the *API* of the iAPI, with
  - $do : C \times A \rightarrow \perp$ being an operation that allows one to *emulate* user actions $a \in A$ on ui;
  - O being a set of *operations* providing iAPI-specific functionality; operations may act locally only (e.g., `getVisibleData`), or they may invoke remote application logic (e.g., `getFullData`); and
  - E being a set of *events* emitted by the iAPI (e.g., in reaction to a navigation action);
– Ctr is the set of *graphical controls* injected into ui to make the features of api accessible interactively.

iAPIs therefore provide programmatic access to both the Surface Web (the graphical UIs) and the Deep Web (the data and logic behind the UIs). The graphical controls injected into the UI bridge between the Deep Web and the Surface Web and make programming interactive. Depending on the *level of support* an iAPI provides to developers, we distinguish ***three types*** of iAPIs:

– *Basic iAPIs* only provide programmatic access limited to the same features a user can perform manually on the UI of the iAPI. That is, $iapi^{base} = \langle ui, \langle do, \emptyset, \emptyset \rangle, \emptyset \rangle$.
– *Intermediate iAPIs* also provide advanced, iAPI-specific operations and events, but still programmatically only. That is, $iapi^{int} = \langle ui, \langle do, O, E \rangle, \emptyset \rangle$, with $O, E \neq \emptyset$.
– *Full iAPIs* provide for interactive programming and enable live, UI-oriented reuse. That is, $iapi^{full} = \langle ui, \langle do, O, E \rangle, Ctr \rangle$, with $O, E, Ctr \neq \emptyset$.

Which kind of iAPI suits best a given reusable application feature depends on the interpretation of the iAPI developer as well as its expected target user. For instance, if the target users are developers, there is no need for graphical controls to manipulate the iAPI; however, if the target users are common web users, graphical controls become mandatory.

Like the model of UIs, also the iAPI model does not prescribe any *level of granularity.* An iAPI may cover a full web page, just like it may be defined only for the table of publications as exemplified in Figure 2. A web page may contain multiple independent iAPIs. It is the developer of the website who decides which features of the site to make accessible for UI-oriented reuse and how. A sensible design will focus on conceptually self-contained features, comprising own UI constructs (e.g., a table or form), application logic and/or data.

3.2 iAPI Use Cases

In order to appreciate the power of this UI-centric perspective on reuse, it is important to understand which new, functional features iAPIs are able to provide to developers. Figure 3 distinguishes four core *iAPI use cases*:

(a) ***Operating UI***: The basic use case is programmatically interacting with a UI, i.e., *operating* it by emulating user interactions, without further reusing any of its features. The use case leverages on the *do* function, which could be programmed by recording and re-playing user interactions.
(b) ***Extracting data***: The second use case is *extracting data* from a UI (e.g., from a table), in order to reuse them inside another piece of software, using own, new UI constructs for the rendering of the data. The use case makes use of data extraction operations in O, whose graphical control could, e.g., simply be dragged/dropped over a new table.
(c) ***Extracting logic***: The more advanced use case is *extracting application logic* from an iAPI, e.g., by processing data via a form and extracting the output from the form's response page or by invoking a remote web service. The use case uses the *do* function to automate form processing as well as the operations and events in O and E.

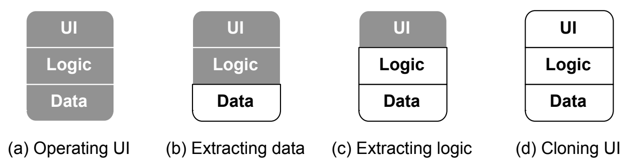


Fig. 3. The four core uses cases of iAPIs for UI-oriented computing. White blocks correspond to what is reused, gray blocks are neglected.

(d) ***Cloning UI***: The most advanced use case is *cloning* a complete piece of UI, along with its underlying application logic and data. This means reconstructing the UI of the iAPI locally and connecting them to the remote iAPI's logic and data. The use case copies *ui* and makes use of all other features, e.g., by dragging/dropping the piece of UI into a new web page.

These four use cases represent a unified and principled solution to the development practices described in the introduction, which otherwise would require mastering a wide set of different component technologies and protocols.

3.3 Implementation Options

Although iAPIs are apparently complex software artifacts themselves, their implementation can be kept relatively simple. Recalling the structure of iAPIs, i.e., $iapi = \langle ui, api, Ctr \rangle$ with $api = \langle do, O, E \rangle$, it is important to note that the *ui* comes essentially for free, in that it is simply a part of a web application's UI, which would be there anyway, with or without the iAPI. Similarly, the function *do* can be provided once for all via a dedicated UI wrapper that enables emulating user interactions with *ui* (similar to Selenium, `http://seleniumhq.org`, but with advanced iAPI support). Only the operations O and the events E require an iAPI-specific implementation, while the graphical controls Ctr can be automatically generated out of their definitions, given a respective rendering convention. As for the operations and events, we identify three options:

– ***Ad hoc implementation***: It is always possible to implement dedicated operations and events via custom JavaScript code included in web pages. Each iAPI would have its own implementation.
– ***iAPI annotations***: If we carefully examined how these implementations look like, we would easily identify recurrent patterns for data extraction from HTML markup, fetching data from a remote source, or invoking remote web services. Instead of implementing these functionalities imperatively in JavaScript, it is possible to factor out the repetitive code into an independent code library and to configure it via declarative annotations of the HTML markup of *ui*. For instance, data extraction can be supported via microformats, while logic extraction may require new annotation elements.

– ***iAPI-ready markup***: Instead of annotating HTML, iAPIs could become an integral part of the HTML standard itself, with own elements to equip UIs natively with interactive programming capabilities and integrated browser support for the rendering of their graphical controls. The idea is similar to the `<header>` and `<navigation>` elements of HTML 5, whose only purpose is to add semantics to the markup and not to actually format content.

Independently of their implementation, the intriguing aspect from a software engineering point of view is that an iAPI's *api* does not exist as independent software artifact. It cannot live without its UI. Both must be instantiated together, and the API's life cycle is tightly coupled to that of its UI counterpart.

This implies some requirements for the construction of new software artifacts (e.g., a web page) out of a set of iAPIs. Specifically, integrating multiple iAPIs requires (i) instantiating the source web pages, (ii) instantiating their iAPIs and (iii) setting up a communication channel among them. The instantiation of the web pages may occur directly inside the web browser or outside in a GUI-less rendering engine, such as HtmlUnit (`http://htmlunit.sourceforge.net`). The instantiation of the iAPIs requires an extension of current rendering engines (e.g., a JavaScript parser and iAPI runtime container). Communications can be set up via interactions among conventional APIs (browser-internal) or web services/sockets (inter-browser) provided by the extensions.

4 A First Implementation

4.1 iAPI Micro-format

The idea to enable the specification of iAPIs is to follow an approach that is similar to that of micro-formats (`http://microformats.org`), with one key difference: while micro-formats provide ready annotations for data types of individual domains (e.g., *hCard* describes people and organizations, and *hCalendar* describes events), the iAPI annotation format provides a set of instructions (i) for the free description of data types of own domains, (ii) for the reuse of existing micro-formats, and (iii) for the reuse of web services or data feeds.

In line with micro-formats, also the iAPI annotation format uses the `class` attribute of HTML to host its annotations (an example follows). Table 1 summarizes the minimum set of instructions identified so far for the reuse of data.

Table 1. Basic iAPI annotations elements

Instruction	Parameter	Description
`iapi`	–	Identifies an iAPI inside HTML
`datafeed`	Feed name	Defines an iAPI for data extraction
`dataitem`	Item name	Identifies individual data items
`dataattribute`	Attribute name	Identifies individual attributes of an item
`rss`	URL	References an equivalent RSS data source
`source`	URL	References a web resource for iAPI reuse
`iapiid`	ID	Identifies an iAPI to be reused inside a source

Given these instructions, a web page that wants to make its data available in the form of an iAPI can be annotated as follows (we call this the *source page*):

```
<ul id="1" class="iapi datafeed:Publications rss:RSS_URL">
  <li class="dataitem:Publication">
     <span class="dataattribute:Author">F. Daniel and A. Furlan</span>.
     <span class="dataattribute:Title">The Interactive API (iAPI)</span>.
     <span class="dataattribute:Event">ComposableWeb 2013</span>
  </li>
  ...
</ul>
```

The annotation identifies the unnumbered list as iAPI (`iapi`), enables the extraction of data (`datafeed`) structured into `dataitem`-s and `dataattribute`-s, specifically of publications with authors, titles, and events. The URL of the `rss` instruction provides an alternative resource where to fetch the same data from.

If a ***developer*** wants to reuse the data exposed by this iAPI (inside a *target page*), he only needs to specify the following line, whose annotation identifies the table as iAPI and tells where to fetch the respective data from (the effect of the interpretation of this annotation format is illustrated in Figure 5):

```
<table class="iapi source:SourceURL iapiid:1"></table>
```

As for now, the annotation format proposes only a set of instructions for the specification of *data extraction* use cases, both from the iAPI's HTML markup as well as from RSS data sources. To limit the annotation effort as much as possible and to foster reuse, inside an `iapi` element it is further possible to use common micro-formats to annotate data, such as the *hCard* and *hCalendar* formats described earlier. The next step will be the design of instructions for the specification of the other iAPI use cases.

4.2 iAPI Middleware

In order to use iAPIs in practice and to enable interactive development, it is necessary to provide for suitable interpreters and runtime environments, i.e., for an *iAPI middleware*. In Figure 4 we illustrate the architecture of an according Chrome browser extension we developed to support the previous annotation example, which acts as middleware. The extension is based on a so-called *background page* for the overall management of the extension and a *content script* for the interaction with the DOM of the page loaded in the browser window. When loading a page, the *iAPI parser* identifies possible iAPIs in the page, and the *HTML augmenter* injects the respective graphical controls. If the identified iAPI reuses data from another source, the HTML augmenter loads (via the *loader* module) the source page, extracts the annotated data, reformats it according to the host HTML element of the target page (e.g., the *Content formatter* turns the unnumbered list into a table), and augments the target page with the newly formatted data. Other parsers support fetching data from different data sources, such as RSS feeds or micro-formats. This is enough to execute the annotations of the source and target pages described above.

With the help of the *Drag&Drop handler*, the extension is also able to intercept drag-and-drop events among browser windows and to allow generic ***web users*** without programming knowledge to drag iAPIs (via the injected graphical controls) from one page to another. This enables the extension to automatically generate the annotation of a target iAPI with information of the source iAPI and to realize the interactive, UI-driven development scenario depicted in Figure 5. Ideally (but this is not implemented yet), the extension would then store the so defined reuse logic either locally or remotely, so as to be able to re-run it as soon as the user returns to the same target page.

5 Benefits and Research Challenges

The aim of this paper is to reconcile the twofold requirement of speeding up development to programmers and of enabling non-programmers to develop. The result are interactive APIs (iAPIs). The approach leverages on the front-end of applications, instead of on their back-end, and proposes a visual, interactive reuse paradigm. Incidentally, these design choices come with some unexpected but highly ***beneficial side-effects***:

- The *deployment* of iAPIs is contextual to the deployment of their host application. iAPIs are an integral part of an application's UI and do not require separate deployment or maintenance. They are natively aligned and consistent with their applications. This is different from web services, which are deployed independently and easily diverge from their applications.
- The *documentation* of iAPIs comes for free to their developers. There is no need for abstract descriptors, IDLs or textual API descriptions. The UI of the page hosting the iAPIs and the graphical controls injected into it already tell everything about the capabilities of the iAPIs contained in the page.

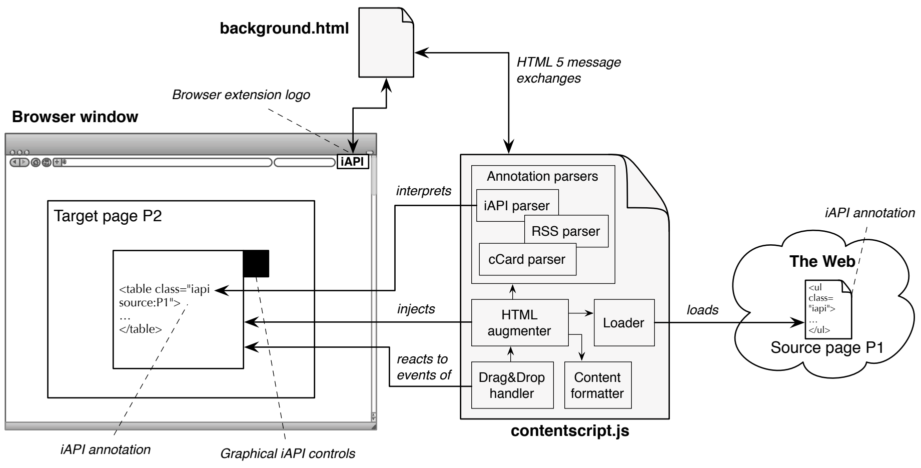
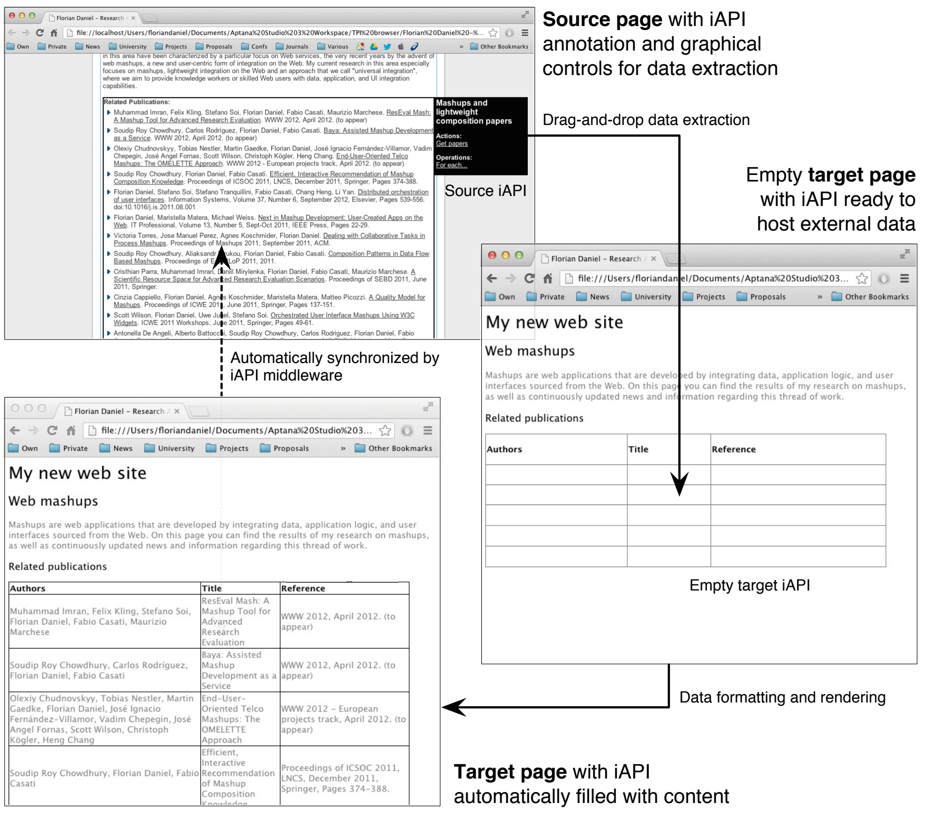


Fig. 4. Architecture of the iAPI middleware Chrome extension


Source page with iAPI annotation and graphical controls for data extraction
Drag-and-drop data extraction
Source iAPI
Empty **target page** with iAPI ready to host external data
Empty target iAPI
Automatically synchronized by iAPI middleware
Data formatting and rendering
Target page with iAPI automatically filled with content
Florian Daniel – Research
Own Private News University Projects Proposals Confs Journals Various Other Bookmarks
Mashups and lightweight composition papers
Actions:
Get papers
Operations:
For each...

file://localhost/Users/floriandaniel/Documents/Aptana%20Studio%203%20Workspace/TPI%20browser/Florian%20Daniel%20-%...

in this area have been characterized by a particular focus on Web services, the very recent years by the advent of web mashups, a new and user-centric form of integration on the Web. My current research in this area especially focuses on mashups, lightweight integration on the Web and an approach that we call "universal integration", where we aim to provide knowledge workers or skilled Web users with data, application, and UI integration capabilities.

Related Publications:

- Muhammad Imran, Felix Kling, Stefano Soi, Florian Daniel, Fabio Casati, Maurizio Marchese. ResEval Mash: A Mashup Tool for Advanced Research Evaluation. WWW 2012, April 2012. (to appear)
- Soudip Roy Chowdhury, Carlos Rodríguez, Florian Daniel, Fabio Casati. Baya: Assisted Mashup Development as a Service. WWW 2012, April 2012. (to appear)
- Olexiy Chudnovskyy, Tobias Nestler, Martin Gaedke, Florian Daniel, José Ignacio Fernández-Villamor, Vadim Chepegin, José Angel Fornas, Scott Wilson, Christoph Kögler, Heng Chang. End-User-Oriented Telco Mashups: The OMELETTE Approach. WWW 2012 - European projects track, April 2012. (to appear)
- Soudip Roy Chowdhury, Florian Daniel, Fabio Casati. Efficient, Interactive Recommendation of Mashup Composition Knowledge. Proceedings of ICSOC 2011, LNCS, December 2011, Springer, Pages 374-388.
- Florian Daniel, Stefano Soi, Stefano Tranquillini, Fabio Casati, Chang Heng, Li Yan. Distributed orchestration of user interfaces. Information Systems, Volume 37, Number 6, September 2012, Elsevier, Pages 539-556. doi:10.1016/j.is.2011.08.001
- Florian Daniel, Maristella Matera, Michael Weiss. Next in Mashup Development: User-Created Apps on the Web. IT Professional, Volume 13, Number 5, Sept-Oct 2011, IEEE Press, Pages 22-29.
- Victoria Torres, Jose Manuel Perez, Agnes Koschmider, Florian Daniel. Dealing with Collaborative Tasks in Process Mashups. Proceedings of Mashups 2011, September 2011, ACM.
- Soudip Roy Chowdhury, Aliaksandr [illegible]ukou, Florian Daniel, Fabio Casati. Composition Patterns in Data Flow Based Mashups. Proceedings of E[illegible]LoP 2011, 2011.
- Cristhian Parra, Muhammad Imran, Daniil Mirylenka, Florian Daniel, Fabio Casati, Maurizio Marchese. A Scientific Resource Space for Advanced Research Evaluation Scenarios. Proceedings of SEBD 2011, June 2011, Springer.
- Cinzia Cappiello, Florian Daniel, Agnes Koschmider, Maristella Matera, Matteo Picozzi. A Quality Model for Mashups. Proceedings of ICWE 2011, June 2011, Springer, Pages 137-151.
- Scott Wilson, Florian Daniel, Uwe Jugel, Stefano Soi. Orchestrated User Interface Mashups Using W3C Widgets. ICWE 2011 Workshops, June 2011, Springer, Pages 49-61.
- Antonella De Angeli, Alberto Battocchi, Soudip Roy Chowdhury, Carlos Rodriguez, Florian Daniel, Fabio

file:///Users/floriandaniel/Documents/Aptana%20Studio%203...

My new web site

Web mashups

Mashups are web applications that are developed by integrating data, application logic, and user interfaces sourced from the Web. On this page you can find the results of my research on mashups, as well as continuously updated news and information regarding this thread of work.

Related publications

Authors	Title	Reference

file:///Users/floriandaniel/Documents/Aptana%20Studio%203...

My new web site

Web mashups

Mashups are web applications that are developed by integrating data, application logic, and user interfaces sourced from the Web. On this page you can find the results of my research on mashups, as well as continuously updated news and information regarding this thread of work.

Related publications

Authors	Title	Reference
Muhammad Imran, Felix Kling, Stefano Soi, Florian Daniel, Fabio Casati, Maurizio Marchese	ResEval Mash: A Mashup Tool for Advanced Research Evaluation	WWW 2012, April 2012. (to appear)
Soudip Roy Chowdhury, Carlos Rodríguez, Florian Daniel, Fabio Casati	Baya: Assisted Mashup Development as a Service	WWW 2012, April 2012. (to appear)
Olexiy Chudnovskyy, Tobias Nestler, Martin Gaedke, Florian Daniel, José Ignacio Fernández-Villamor, Vadim Chepegin, José Angel Fornas, Scott Wilson, Christoph Kögler, Heng Chang	End-User-Oriented Telco Mashups: The OMELETTE Approach	WWW 2012 – European projects track, April 2012. (to appear)
Soudip Roy Chowdhury, Florian Daniel, Fabio Casati	Efficient, Interactive Recommendation of Mashup Composition Knowledge	Proceedings of ICSOC 2011, LNCS, December 2011, Springer, Pages 374–388.

Fig. 5. The concept of interactive, UI-oriented development explained with an example (a video of the flow is available on-line via `http://goo.gl/bAzS2`)

- The *retrieval* of iAPIs does not ask for new infrastructure or query paradigms. Being iAPIs an integral part of the Surface Web, it is enough to query for a desired functionality via common web search. If a suitable iAPI exists, its web page will pop up under the search results. Again, this is different from web services, whose registry infrastructure (UDDI) is a well-known failure.
- Finally, *understanding* the logic of an iAPI does not require programming skills or computer science knowledge. Common web browsing skills are enough to understand the available features and how to operate them.

These properties make iAPIs a technology that is intrinsically *Web-ready* and that has the potential to *boost reuse* on the Web. Yet, in order for this to happen, a set of ***research and engineering challenges*** ask for suitable answers:

- *iAPI development*: It is crucial to sensibly conceive how iAPIs are developed, in order to keep their development sustainable. An interpretable annotation format (as exemplified in Section 4) or markup language seems promising, but this is not the only option.

– *iAPI extraction*: In order to create a critical mass of iAPIs, it may be necessary to "extract" iAPIs from existing web applications, similar to web data extraction, e.g., by externally annotating third-party web applications.
– *UI-oriented computing infrastructure*: It is necessary to complement iAPIs with suitable UI-oriented middleware, runtime environments, communication protocols, browser extensions, and similar.
– *Interactive, live programming*: It is an HCI challenge to design an effective UI-oriented programming paradigm for non-programmers.

In summary, iAPIs aim to set the agenda for a new line of research, i.e., ***UI-oriented computing***, which brings together SE/WE and HCI in a completely new fashion. Compared to micro-formats or Semantic Web approaches, the bet of iAPIs is that turning UIs into programming artifacts will also lead to support from developers and content providers. The idea to do so is to equip UIs with semantics (via annotations), which do not only identify reusable UI elements inside an page, but also tell how to reuse them. This is fundamentally different from the approach, for example, described in [6], where the authors study how to reverse-engineer Java-based UIs from Java code by monitoring UI usage via aspect-oriented extensions of the source code of applications, in essence demonstrating that there is a lack of semantics and support for reuse of UIs.

The implementation described in this paper is as proof of concept and a starting point for future development. Concretely, via the *W3C Interactive APIs Community Group* (`http://www.w3.org/community/interative-apis`) we aim to develop a full-fledged iAPI annotation format with the help of the community (participation is open and free). On `http://www.interactive-apis.org` we would like to host the open-source projects for iAPI middleware.

References

1. Abdelnur, A., Hepper, S.: Java Portlet Specification, Version 1.0. Technical Report JSR 168, Sun Microsystems, Inc. (October 2003), `http://download.oracle.com/otndocs/jcp/PORTLET_1.0-FR-SPEC-G-F/`
2. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services: Concepts, Architectures, and Applications. Springer (2003)
3. Dix, A., Finlay, J., Abowd, G., Beale, R.: Human-Computer Interaction, 3rd edn. Prentice Hall (2004)
4. Fielding, R.: Architectural Styles and the Design of Network-based Software Architectures. Ph.d. dissertation, University of California, Irvine (2007)
5. Jordan, D., Evdemon, J.: Web Services Business Process Execution Language Version 2.0. Oasis standard, OASIS (April 2007), `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`
6. Li, P., Wohlstadter, E.: View-based maintenance of graphical user interfaces. In: AOSD, pp. 156–167 (2008)
7. Namoun, A., Nestler, T., Angeli, A.D.: Service Composition for Non-programmers: Prospects, Problems, and Design Recommendations. In: ECOWS, pp. 123–130 (2010)

8. Nottingham, M., Sayre, R.: The Atom Syndication Format (December 2005), `http://www.ietf.org/rfc/rfc4287.txt`
9. RSS Advisory Board. RSS 2.0 Specification (2009), `http://www.rssboard.org/rss-specification`
10. Web Application Working Group. Widgets Family of Specifications (May 2012)
11. Yu, J., Benatallah, B., Casati, F., Daniel, F.: Understanding Mashup Development. IEEE Internet Computing 12(5), 44–52 (2008)
12. Yu, J., Benatallah, B., Saint-Paul, R., Casati, F., Daniel, F., Matera, M.: A Framework for Rapid Integration of Presentation Components. In: WWW, pp. 923–932. ACM Press (May 2007)