# Table-Driven Customization of Web Applications

Anonymous Author(s)

## Abstract

In this paper we propose *table-driven customization,* a new paradigm for enabling end users to customize software applications without doing traditional programming. Users directly manipulate a tabular view of the structured data inside the application, rather than writing imperative scripts as in most customization tools. We extend this simple model with a spreadsheet formula language and custom data editing widgets, which provide sufficient expressivity to implement many useful customizations.

We describe Wildcard, a browser extension which demonstrates table-driven customization in the context of web applications. Through concrete examples, we demonstrate that this paradigm can be used to create useful customizations for a variety of real applications. We share reflections from our usage of the Wildcard system, including its strengths and limitations relative to other customization approaches. We further explore how new software architectures might help application developers promote this style of end-user customization.

***CCS Concepts*** • **Software and its engineering → Integrated and visual development environments**.

***Keywords*** end-user programming, software customization, web browser extensions

**ACM Reference Format:**
Anonymous Author(s). 2020. Table-Driven Customization of Web Applications. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

Many software applications don't meet the precise needs of their users. When this happens, the only recourse is to complain to the developers, or, more likely, to simply give up. Back in 1977, in Personal Dynamic Media [**?** ], Alan Kay envisioned personal computing as a medium that let a user "mold and channel its power to his own needs," but today, software behaves more like concrete than clay.

- many approaches to customizing software. Most are imperative command based. Can't get around it.
- there are successful systems for visual SQL queries: Airtable, Sieuferd. Even spreadsheets.
- what if you could see the data queries powering a website and directly modify them yourself?
- we call this table-driven customization.

To make it work with the real world, we develop Wildcard, shim this on top of existing web apps. Show in Section Examples that it really works! can do tons of useful stuff with real apps.

Architecture: Describe implementation of Wildcard. We've developed generic abstractions that make this system even more extensible in the future. we also propose a future architecture where apps are built for this; they expose their data queries and their structured data results.

Design Principles: explain ideas behind the system, generalizable to software customization more generally

Related Work: table editors,

## 2 Examples

*This section still todo. Borrow heavily from Convivial paper, but extend with newer results*

- Just use airbnb + expedia from Convivial paper?
- instacart + hacker news?
- summarize/mention all 12 sites we've used it with
- real usage:
  - 1 externally contributed adapter
  - flux adapter in actual use by Sergio

## 3 System architecture

The basic data unit in table-driven customization is a Table: an ordered list of tuples, each of which has an ID and some attributes. The user always sees a single table in the table editor, but this table is the result of combining data across one or more underlying tables, much like viewing the result of a SQL query across multiple database tables. Each individual table's data is managed by a "table adapter" component, and a query engine coordinates across the various table adapters.

(*Todo: drive this explanation via concrete Instacart example. Add a figure.*) For example, one table adapter manages scraping data from the DOM of a site, while another manages extracting data from AJAX requests. Each table adapter produces a table of data, and the two are joined together by a shared ID column to produce what looks like a single table of data extracted from the site. Similarly, when the user

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*Conference'17, July 2017, Washington, DC, USA*
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

makes annotations on a site, these annotations are handled by a third table adapter, which manages a data table stored in the browser's local storage.

### 3.1 Table adapter API

In order to promote uniformity within the system, we've defined a small abstract API, which all table adapters conform to. (Todo: formalize this more, make it more specific?)

*Output: data table:*The main purpose of a table adapter is to return a table representing some data source. The table is live-updating: a table adapter can produce a new version of its data table at any point (e.g., updating the data table in response to a change in a website's DOM). When this happens the new data is pushed to the editor and the query results are refreshed.

Table editors also accept various incoming commands.

*Input: record edits:*The editor can request to a table adapter to make an edit to a record. The meaning of making an edit can vary depending on the adapter: in the local storage adapter, an annotation can be persisted into local storage; in the DOM scraping adapter, an edit can represent filling in a form field. An adapter can also mark values as read-only if it wouldn't be meaningful to edit them; for example, the DOM scraping adapter typically marks page content as read-only, except for editable form fields.

*Input: query information:*The query engine also sends each table adapter information about the entire cross-table query view being created by the user. This functionality is currently mainly used by the DOM scraping adapter to make changes to the original web page:

- observes the sort order of the query view to re-order rows in the page
- observes the content of other joined tables to render annotations in the page
- observes the currently selected row in the UI, to highlight the corresponding row in the page

### 3.2 Table adapters

Here we describe the three types of table adapters which we have implemented so far to power the customizations shown above. Then, to demonstrate the generality of the table adapter paradigm, we describe three more hypothetical types of table adapters which could extend the power of the Wildcard system.

#### 3.2.1 Existing Adapters

A **DOM scraping adapter** extracts data from the DOM of a web page, and manipulates the DOM to re-order rows, edit form entries, and inject annotations. A programmer only needs to write a single function which returns scraped data and pointers to relevant DOM elements; the Wildcard framework uses this function to implement the rest of the needed functionality.

(*do we want to call DOM scraping adapter a "live" adapter because its data is also shown elsewhere…?*)

An **AJAX scraping adapter** intercepts AJAX requests made by a web page, and extracts information from those requests. A programmer writes a function which specifies how to extract data from an AJAX request, and the framework handles the details of actually intercepting requests and calling the programmer-defined function.

The **local storage adapter** simply stores a table of data in the browser.

#### 3.2.2 Future Adapters

we have intentionally designed the table adapter API to be general enough to encapsulate other types of data and additional functionality in the future. Here are three concrete examples of such possibilities:

**Integrated website adapters**: We have taken pains to design a customization system which is not limited to web scraping as the only means for integrating with existing sites, and can also accommodate first party developers adding support for table-driven customization directly into their own websites. A DOM Scraping table adapter could be swapped out for such an "integrated website adapter," which implements the exact same interface as a scraping adapter but by directly accessing the internal state of the user interface, without needing any scraping logic.

While we have not yet created a fully operational integrated website adapter, we think it is possible to create such adapters for existing frontend web frameworks which result in minimal effort on the part of the application developers. For example, we have created an early prototype of a plugin for the Redux state management library, which uses the Model-View-Update pattern that represents the entire state of a user interface as a single centralized object. To configure an integrated website adapter for such an existing application, the user can specify a function projecting the centralized application state as a table, and handlers for how data edits to the table should affect the state.

**Shared storage adapter**: It would be useful to share user annotations between people and across devices—for example, collaboratively taking notes with friends on a list of options for places to stay on Airbnb. The existing Local Storage Adapter could be extended to share live synchronized data with other users. This could be achieved through a centralized web server (perhaps an existing service like Google Sheets capable of storing tabular data), or through P2P connections which might improve the privacy guarantees available for the shared annotations.

**Third party API data adapter**: Currently, the main mechanism for including data from web APIs in Wildcard is using spreadsheet formulas. However, a web API could also be wrapped to expose a table API that would dynamically create tables in response to queries. For example, when fetching walkability scores for many GPS locations, the query engine

could request a table of walkability scores for various pairs of latitude and longitude, and the table adapter could dynamically perform API queries to populate a result table. (*todo: why is this better than spreadsheet formulas? I don't think it offers any more room for batching performance optimization because we've already framed formulas as whole-column, not row specific*)

### 3.3 Query engine

The query engine is responsible for coordinating across multiple table adapters. It joins data across multiple tables and creates a single result table which is shown to the user through the editor. It also handles all user interactions and routes appropriate messages to each table adapter.

First, every query involves a primary DOM scraping table adapter which associates records in the result with elements in the application's user interface. At minimum, the primary table adapter needs to return record IDs and have the ability to manipulate the application's UI. It can also optionally return data about each record.

Next, additional tables (AJAX data, local storage data) are left joined by ID. (*todo: discuss IDs here?*) Finally, the result table can be sorted and filtered by any column.

One way to think of this model is a tiny constrained subset of the SQL query model. We've found that this simple model has proven sufficient for meeting the needs of customization in practice, and minimizes the complexity of supporting more general and arbitrary queries. But because it fits into the general SQL paradigm, it could theoretically be extended to support more types of queries.

- formulas: do they go here?

### 3.4 Table editor

- Our editor is closer to visual SQL query engines like Sieuferd or Airtable than a freeform spreadsheet.
- cell editors
- Note that there could be other entire table editors

## 4 Design aspects

### 4.1 Data-oriented programming

There have been many attempts at simplifying aspects of traditional programming to enable non-programmer end users to perform customization. End user scripting languages (AppleScript, VBA, Chickenfoot, CoScripter) have innovated at the syntax level, aiming to improve ease-of-use. Visual programming environments (Mac Automator, Zapier) go further by eliminating text syntax entirely, but maintain the same computational model. Programming-by-demonstration and macro recording environments (Applescript, Helena, WebVCR) remove some of the initial programming burden by starting with concrete demonstrations, but typically require programming later in the process.

However, all of these approaches share something in common: **an imperative programming model**. They all involve a list of commands executed in sequence, with control flow constructs like conditionals and loops, and mutable variables. This makes some intuitive sense: many customizations seem most naturally expressed as a series of commands, each of which has a side effect on the UI: first enter this value, then click this button, etc. The result is that end users eventually need to become familiar with all of these traditional imperative programming constructs, even if only in diluted form. (todo: bolster the case that this is a barrier to learning)

Imperative programming is not the only possible model. Spreadsheets have demonstrated another computational model: data editing via direct manipulation, combined with pure functional expressions for computation, with automatic reactive updates provided by the runtime. We call this a data-oriented model, because the user is focused on editing data values and producing the correct derived data values, rather than encoding a sequence of commands. The success of spreadsheets with end user programmers suggests that finding a way to apply data-oriented programming model to software customization might enable more people to modify their software. However, it's not immediately obvious how the spreadsheet paradigm applies to customization.

A key insight from our work is that, rather than representing customization as a series of commands issued to an existing application, we can present a projected view of the application's internal state, and let the user directly edit that projection as a means of customizing the application. In some sense, this is similar to the DOM inspector, which provides a user with an alternate representation of the UI that they can directly manipulate (e.g., by selecting an element and deleting it from the DOM tree). But rather than provide a low-level view of the user interface elements, we instead provide a higher-level view of the semantic state of the application.

### 4.2 Black-box + semantic

Software customization tools typically fall into one of two categories: black-box, or semantic.

**Black-box customization tools** enable customization without using official extension APIs, enabling a broader range of customizations on top of more applications. For example, web browser extensions have demonstrated the utility of customizing websites through manipulating the DOM, without needing explicit extension APIs to be built in. However, black-box customization comes with a corresponding cost: these tools can typically only operate at a low level of abstraction, e.g. manipulating user interface elements. This makes it harder for end users to write scripts, and makes the resulting scripts more brittle. (todo: support this claim more, provide examples)

**Semantic customization tools** use explicit extension APIs provided by the application developer. Examples of this

include accessing a backend web API, or writing a customization in Applescript that uses an application-specific API. The main benefit is that this allows the extension author to work with meaningful concepts in the application domain—"create a new calendar event" rather than "click the button that says new event."—which makes customizations easier to build and more robust. However, this style limits the range of extensions that can be built, to only those which the official plugin API supports.

(Footnote this paragraph?) This categorization is slightly oversimplified, and there are existing customization ecosystems that span both categories. For example, part of the success of browser extensions stems from the fact that the DOM encourages the use of standardized semantic UI elements; even if an app developer doesn't anticipate customization, merely using of semantic and accessible HTML creates a collateral benefit of easier extension. Another example of a middle ground is AppleScript, which provides system-wide low-level APIs for manipulating GUIs as a means of black-box customization, while also enabling developers to optionally add application-specific APIs to add support for semantic customization.

With Wildcard, we use a hybrid approach that takes the best of both worlds. Programmers implement an API wrapper that is internally implemented as a black-box customization, but externally provides a semantic interface to the application. End users get an ergonomic and simplified customization experience, but without the need to depend on a first-party application developer exposing extension APIs.

One way to view this approach is as introducing a new abstraction barrier into black-box extension. Typically, a black box customization script combines two responsibilities: 1) mapping the low-level details of a user interface to semantic constructs (e.g., using CSS selectors to find certain page elements), and 2) the actual logic of the specific customization. (todo: could easily show examples of this from browser extensions, Chickenfoot, etc) Even though the mapping logic is often more generic than the specific customization (e.g., finding a given input element is independent of what text to insert into that element), the intertwining of these two responsibilities in a single script makes it very difficult to share the mapping logic across scripts. With Wildcard we propose a decoupling of these two layers: a community-maintained mapping layer, shared across many specific customizations by individual users. This architecture has been successfully used by projects like Gmail.js, an open source project that creates a convenient API for browser extensions to interface with the Gmail web email client.

### 4.3 Proximity to use

*borrow from convivial paper's discussion of in-place*

- here's where "automation vs customization" comes in: automation frames it as entering a whole new scripting environment; we frame it more as just an alternate UI that you can use.
- this is the heart of the gentle slope
- backend APIs: terrible
- the best would be right inline with the GUI (cite Scotty, instrumental work) but this has its own problems
- we settle for a compromise: an alternate structured view. (explicitly contrast this clearly)
- the CLI GUI thing

## 5 Related Work

Find a way to organize all this:

- our own workshop paper. since then…
  - fundamentally rearchitected around table adapters
  - evaluated on many more websites
  - more fully describing how the system works
- browser extensions
- instrumental interaction, Scotty, Webstrates
- customization research tools: Chickenfoot, Coscripter
  - Wildcard, like Chickenfoot, wants to hide HTML from users. But we show a structured data view, whereas Chickenfoot shows nothing
- desktop customization: Applescript, VBA, COM
- browser dev tools
- database GUIs
- spreadsheets