Customizing Software by Direct Manipulation of Structured Data

Anonymous Author(s)

Abstract

In this paper we show how the behavior of a software application can be extended and adapted by direct manipulation, using table-driven customization, a new paradigm that allows end users to customize applications without writing traditional code.

Instead, users directly manipulate a tabular view of the structured data inside the application—rather than writing imperative scripts (as in most customization tools). This simple model also accommodates a spreadsheet formula language and custom data editing widgets, which provide sufficient expressivity to implement many useful customizations.

We illustrate the approach with Wildcard, a browser extension that implements table-driven customization in the context of web applications. Through concrete examples, we show that our paradigm can be used to create useful customizations for real applications. We share reflections from experience using the Wildcard system, on both its strengths and limitations relative to other customization approaches. Finally, we explore how this paradigm might lead to new software architectures that encourage this form of 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. Customizing Software by Direct Manipulation of Structured Data. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Most attempts at empowering end users to customize their software offer a simplified version of programming. Some scripting languages (AppleScript, Chickenfoot) have a friendly syntax that resembles natural language. Visual programming tools (Mac Automator, Zapier) eliminate text syntax entirely. Macro recorders (Applescript, Helena, WebVCR) remove some of the initial programming burden by letting a user start with concrete demonstrations.

Despite their many differences, these approaches all share something in common: an imperative programming model, with statement sequencing, mutable variables and loops. End users express their ideas in scripts—sequences of commands—which, name aside, are not very different from conventional code.

We have known for decades about an alternate approach: *direct manipulation* [11], where "visibility of the object of interest" replaces "complex command language syntax". Direct manipulation is the *de facto* standard in GUIs today, but when it comes to customizing software, it is rarely to be found. In this work, we ask: what would it look like to build a software customization interface that relies on direct manipulation? We take inspiration from spreadsheets and visual database query interfaces [1, 3], which have successfully enabled end users to run queries and computations through direct manipulation of data.

In this paper we present a technique called *table-driven customization*, which applies ideas from visual query interfaces in the context of software customization. An application's UI is augmented with a table view, where the user can see and manipulate the application's internal data . Changes in the table view result in immediate corresponding changes to the original user interface of the application, enabling the user to customize an application with live feedback.

We have developed a browser extension called Wildcard which uses web scraping techniques to implement table-driven customization for existing Web applications. In Section 2, we introduce the key ideas of table-driven customization by presenting several examples of real customizations implemented in Wildcard.

In Section 3, we explain the architecture of table-driven customization. We focus on the *table adapter* abstraction, which allows many different types of underlying data to be bidirectionally mapped to a table. We describe several types of table adapters we've built in Wildcard, and also describe future adapters that are supported by the general paradigm.

We have used Wildcard to build real customizations for 11 different websites. In Section 5, we present reflections from this process, outlining the kinds of customizations we were

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

able to build, limitations we encountered, and reflections on the ease of integrating scraping logic with real websites.

In Section 4, we discuss some key themes from our work:

- *Customization by direct manipulation*: End users should be able to customize an application by directly examining and modifying its data, rather than by writing imperative scripts.
- *Wrapping applications for customization*: Typically, tools that don't rely on official extension APIs resort to offering low-level APIs for customization. Instead, we propose a community-maintained library of semantic wrappers around existing applications, enabling end users to work with domain objects rather than low-level representations.

Table-driven customization relates to existing work in many areas. In particular, our goals overlap with many software customization tools, and our methods overlap with direct manipulation interfaces for working with structured data, including visual database query systems and spreadsheets. We explore these connections and more in Section 6.

2 Examples

todo: maybe redo this as a separate example to avoid self plagiarism?

To concretely illustrate the end user experience of table-driven customization, here are several real examples of using the Wildcard browser extension to customize websites.

2.1 Augmenting search results

In 2012, the travel site Airbnb removed the ability to sort accommodation searches by price. Users could still filter by price range, but could no longer view the cheapest listings first. Many users complained that the change seemed hostile to users. "It's so frustrating! What is the logic behind not having this function?" said one user on the Airbnb support forum. Alas, the feature remains missing to this day.

Using Wildcard, the user can fix this omission, while leaving the page's design and the rest of its functionality unchanged. Figure 1 shows an overview of augmenting the Airbnb site. First, the user opens up the Wildcard panel, which shows a table corresponding to the search results in the page. As they click around in the table, the corresponding row in the page is highlighted to indicate the mapping between the views.

Then, the user clicks on the price column header to sort the spreadsheet and the Airbnb UI by price (Figure 1, Note A). They also filter to listings with a user rating above 4.5 (another feature missing in the original Airbnb UI).

After manipulating the data, the user closes the table view and continues using the website. Because the application's UI usually has a nicer visual design than a spreadsheet, Wildcard does not aim to replace it—at any time, the user can use either the UI, the spreadsheet, or both together.

Many websites that show lists of data also offer actions on rows in the table, like adding an item to a shopping cart. Wildcard has the ability to make these "row actions" available in the data table through the site adapter. In the Airbnb UI, saving multiple listings to a Favorites list requires tediously clicking through them one by one. Using Wildcard row actions, the user can select multiple rows and favorite all of them with a single click (Figure 1, Note B). Similarly, the user can open the detailed pages for many listings at once.

Next, the user wants to jot down some notes about each listing. To do this, they type some notes into an additional column in each row, and the notes appear inside the listings in the original UI (Figure 1, Note C). The annotations are saved in the browser and associated with the backend ID of the listing, so they will appear in future browser sessions that display the same listing.

Wildcard also includes a formula language that enables more sophisticated customizations. When traveling without a car, it's useful to evaluate potential places to stay based on how walkable the surroundings are. Using a formula, the user can integrate Airbnb with Walkscore, an API that rates the walkability of any location on a 1-100 scale. When the user calls the `walkscore` formula with the listing's latitude and longitude as arguments, it returns the walk score for that location and shows it as the cell value. Because the cell's contents are injected into the page, the score also appears in the UI (Figure 1, Note D).

2.2 Snoozing todos

In addition to fetching data from other sources, Wildcard formulas can also perform computations. In this example, the user would like to augment the TodoMVC todo list app with a "snooze" feature, which will temporarily hide a todo from the list until a certain date. Figure 2 shows an overview of this customization.

The user opens the table view, which shows the text and completed status of each todo. They start the customization by adding a new column to store the snooze date for each todo (Figure 2, Note A).

The next step is to hide snoozed todos. The user creates a `snoozed?` column, which uses a formula to compute whether a todo is snoozed—i.e., whether it has a snooze date in the future (Figure 2, Note B). Then, they simply filter the table to hide the snoozed todos (Figure 2, Note C).

Because the built-in `NOW()` function always returns the current datetime, snoozed todos will automatically appear once their snooze date arrives.

Because this implementation of snoozing was built on the spreadsheet abstraction, it is completely decoupled from this particular todo list app. We envision that users could share these types of customizations as generic browser extensions, which could be applied to any site supported by Wildcard with no additional effort.

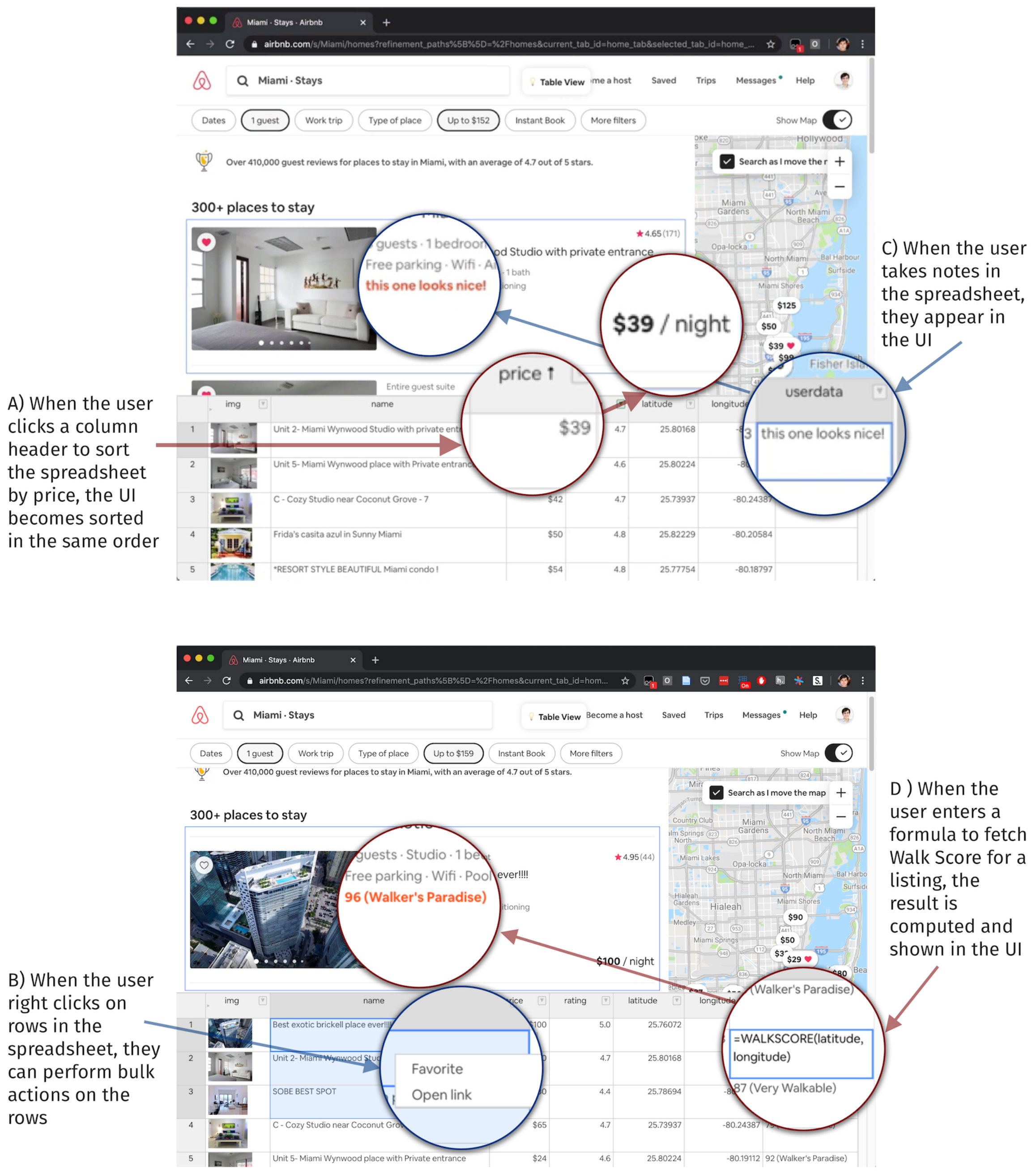
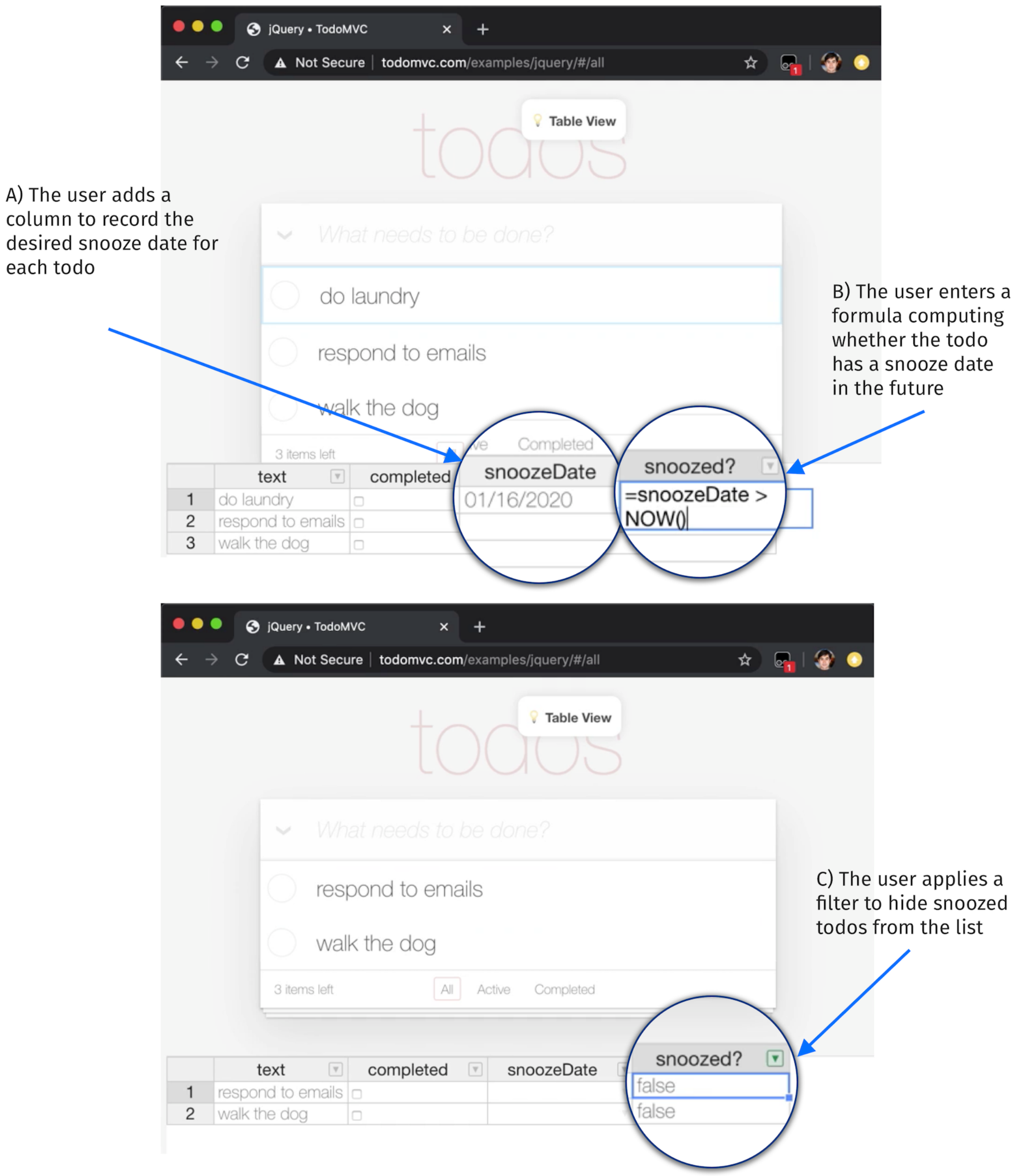

A) When the user clicks a column header to sort the spreadsheet by price, the UI becomes sorted in the same order
C) When the user takes notes in the spreadsheet, they appear in the UI
this one looks nice!
$39 / night
price ↑
userdata
B) When the user right clicks on rows in the spreadsheet, they can perform bulk actions on the rows
Favorite
Open link
D) When the user enters a formula to fetch Walk Score for a listing, the result is computed and shown in the UI
96 (Walker's Paradise)
=WALKSCORE(latitude, longitude)
87 (Very Walkable)

Figure 1. Using Wildcard to augment the Airbnb search page for booking accommodations


jQuery • TodoMVC
Not Secure | todomvc.com/examples/jquery/#/all
Table View
todos
What needs to be done?
do laundry
respond to emails
walk the dog
3 items left
All
Active
Completed
A) The user adds a column to record the desired snooze date for each todo
B) The user enters a formula computing whether the todo has a snooze date in the future
C) The user applies a filter to hide snoozed todos from the list

	text	completed	snoozeDate	snoozed?
1	do laundry	☐	01/16/2020	=snoozeDate > NOW()
2	respond to emails	☐		
3	walk the dog	☐		

	text	completed	snoozeDate	snoozed?
1	respond to emails	☐		false
2	walk the dog	☐		false

Figure 2. Using Wildcard to add a "snooze" feature to the TodoMVC todo list app

2.3 Adding a custom datepicker

It might seem that Wildcard is only useful on websites that display lists of tabular data, but the table metaphor is flexible enough to represent many types of data. For example, a flight search form on Expedia can be represented as a single row, with a column corresponding to each input (Figure 3, Note A).

In some of the previous examples, the table cells were read-only (because users can't, for example, change the name or price of an Airbnb listing). In this case, the cells are writable, which means that changes in the table are reflected in the form inputs. This becomes especially useful when combined with GUI widgets that can edit the value of a table cell.

Filling in dates for a flight search often requires a cumbersome workflow: open up a separate calendar app, find the dates for the trip, and then carefully copy them into the form. In Wildcard, the user can avoid this by using a datepicker widget that shows the user's personal calendar events (Figure 3, Note B). The user can directly click on the correct

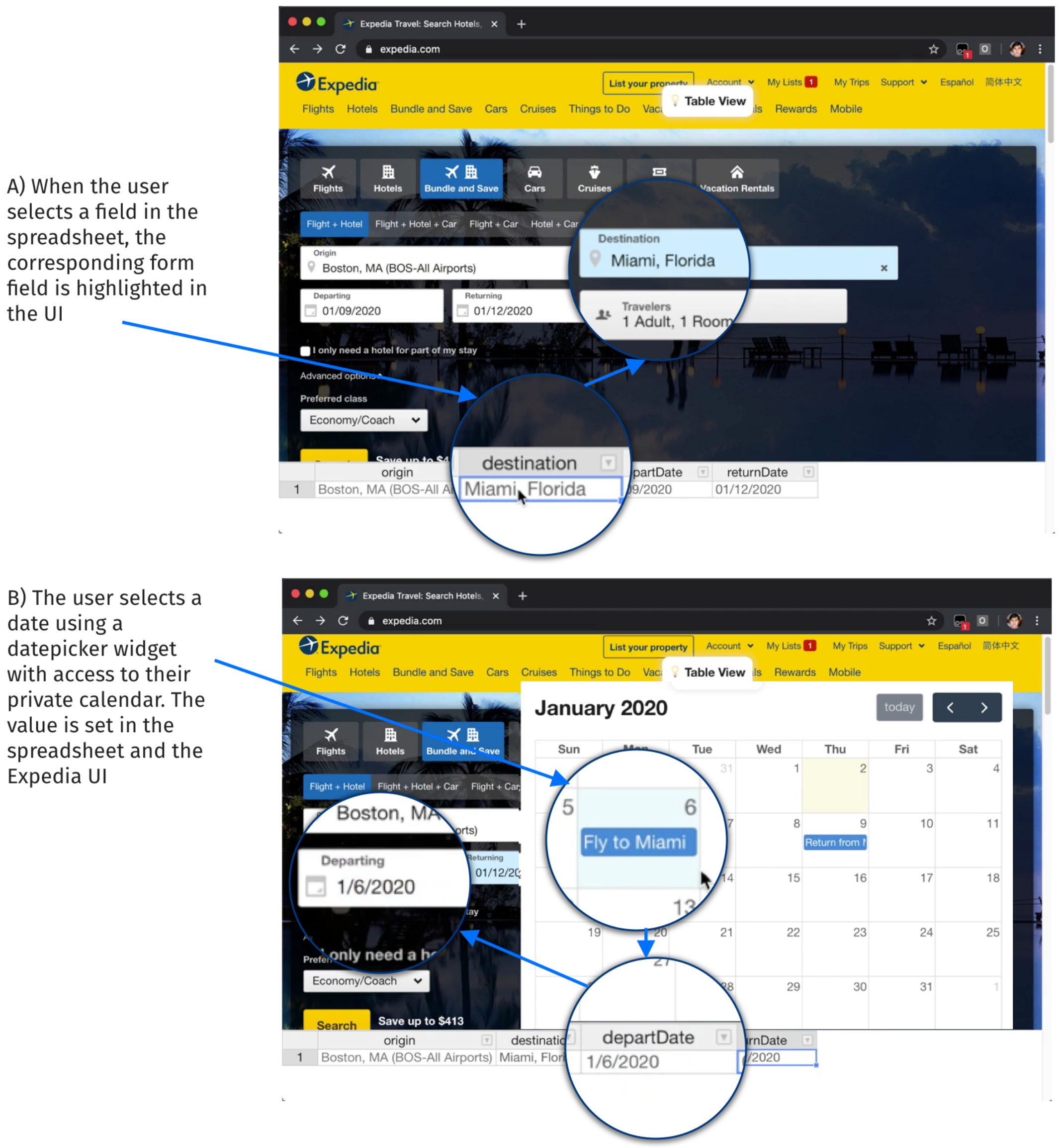

A) When the user selects a field in the spreadsheet, the corresponding form field is highlighted in the UI
B) The user selects a date using a datepicker widget with access to their private calendar. The value is set in the spreadsheet and the Expedia UI
Table View
Destination
Miami, Florida
destination
Miami, Florida
January 2020
Fly to Miami
Departing
1/6/2020
departDate
1/6/2020

Figure 3. Using Wildcard to augment the Expedia page for booking a flight

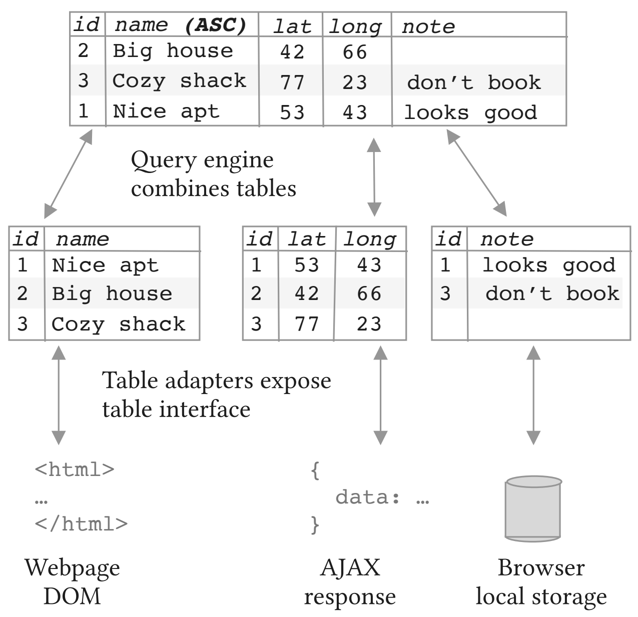
date, and it gets inserted into both the spreadsheet and the original form.n

3 System architecture

Figure 4 summarizes the overall architecture of table-driven customization, using a simplified version of the Airbnb example above. In this example, the name of each listing is scraped from the web page DOM, the latitude and longitude of each listing is scraped from AJAX responses, and user annotations are loaded from the brower's local storage.

First, the three different data sources are each bidirectionally mapped to a table interface by a **table adapter**. The table adapter defines how to map a particular type of data to a table, and what effects edits should have on the original data source. In some cases, the mapping logic is straightforward: the local storage adapter stores a table of data, so the mapping to the table abstraction is trivial. In other cases, the mapping is more involved: the DOM scraping adapter implements web scraping logic to produce a table of data from the web page, and turns edits to the table into DOM manipulations like reordering rows of data on the page.

The three separate tables are then combined into a single table for the end user to view and edit. The **query engine** is responsible for creating this combined view, and routing the user's edits back to the individual table adapters. In this example, the query engine has joined the three tables together



id	name (ASC)	lat	long	note
2	Big house	42	66	
3	Cozy shack	77	23	don't book
1	Nice apt	53	43	looks good

id	name
1	Nice apt
2	Big house
3	Cozy shack

id	lat	long
1	53	43
2	42	66
3	77	23

id	note
1	looks good
3	don't book

```
<html>
…
</html>
```

```
{
  data: …
}
```

Query engine combines tables
Table adapters expose table interface
Webpage DOM
AJAX response
Browser local storage

Figure 4. The table adapter architecture

by a shared ID column, and sorted the result by the name column.

We now examine each component of the system in more detail.

3.1 Table adapters

A key idea in table-driven customization is that a wide variety of data sources can be mapped to the generic abstraction of an editable table. In a relational database, the table matches the underlying storage format, but in table-driven customization, the table is merely an *interface layer*. The data shown in the table is a projection of some underlying state, and edits to the table can have complex effects on the underlying state.

Externally, a table adapter must satisfy an abstract interface. The first two parts of the interface resemble the interface exposed by a table in a relational database:

Returning a table: A table adapter exposes a table of data: an ordered list of records. Each record carries a unique identifier and associates named attributes with values. Tables have a typed schema, so the same attributes are shared across all records. A table adapter can update the contents of a table at any time, in response to changes in the underlying state (e.g., a DOM scraping adapter can update the table when the page body changes). When data changes, the query view is reactively updated in response.

Making edits: The query engine can issue a 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 may be persisted into local storage; in the DOM scraping adapter, an edit may 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.

The query engine also sends additional information about the combined query view to each table adapter. These functions are currently used to provide the DOM scraping adapter with sufficient information to manipulate the original application UI as the user manipulates the table view.

Sorting/filtering: When the user sorts or filters the query view, an ordered list of visible IDs is sent to each table adapter. The DOM scraping adapter uses this information to modify the list of visible rows in the web page.

Information from other tables: The query engine sends each table adapter the entire combined table of data being rendered to the user. The DOM scraping adapter uses this to add annotations to the page by looking for additional data columns joined to scraped rows, and rendering them in the page.

Currently selected record: The query engine notifies each table adapter about which record is currently selected by the user in the table UI. The DOM scraping adapter uses this information to highlight the row in the page that corresponds to the selected record in the table.

3.1.1 Types of table adapters

Here we describe in more detail the table adapters that we have implemented in Wildcard to power the customizations shown in Section 2.

DOM scraping adapters are the essential component that enables Wildcard to interface with an existing website UI. In addition to the standard web scraping problem of extracting a table of data from the DOM, a scraping adapter must also manipulate the DOM to reorder rows, edit form entries, and inject annotations as the table is edited.

Because each website has unique content, we rely on programmers to create a DOM scraping adapter for each individual website to make it available for customization in Wildcard. To make this approach viable, we have built a library of functions that make it easy to implement a scraping adapter, requiring the programmer only to implement the minimal site-specific parts. The programmer writes a single Javascript function which can use standard scraping techniques like CSS selectors and DOM APIs to extract relevant elements from the page. The library functions then wrap this function to implement the rest of the needed functionality; for example, when the table is sorted, we find the DOM elements corresponding to the rows in the table, remove them from the page, and then reinsert them in the new order.

An **AJAX scraping adapter** intercepts AJAX requests made by a web page, and extracts information from those requests. When available, this tends to be a helpful technique because the data is already in a structured form, and often includes information not shown in the UI. As with DOM scraping adapters, we have made it easy for programmers to create site-specific AJAX scraping adapters. A programmer writes a function that specifies how to extract data from an AJAX request, and the framework handles the details of intercepting requests and calling the programmer-defined function.[1]

The **local storage adapter** simply stores a table of data in the browser. As shown in the example use cases, this can be useful for persisting annotations in a private way, without uploading them to a web service.

3.1.2 Future Adapters

We have designed the table adapter API to be general enough to support other types of useful adapters in the future. Here are two such possibilities:

Integrated website adapters: A key benefit of the table adapter abstraction is that Wildcard is not coupled to web scraping as the only means for integrating with existing sites, but can also accommodate first party developers adding support directly into their own websites. An "integrated website adapter" installed by the developer could directly accesses the internal state of the application, providing the same functionality as a DOM scraping adapter but in a more robust way.

With the advent of rich frontend web frameworks, structured application state is now often available in the web client. We suspect it is possible to create plugins for frontend frameworks that expose this state to Wildcard with only minimal effort from the application developers. To test this hypothesis, we have created an early prototype of a Wildcard adapter for the Redux state management library, and used it with the TodoMVC todo list application. To install the adapter, the programmer only needs to make a few small changes to their app:

Exposing a table: In Redux, app developers already represent the state of a user interface as a single object. To expose a table of data, the developer simply writes a function that maps the state object to a table. *Editing data values*: The developer defines how edits to records in the table should map to edits in the state object. *Sorting/filtering the UI*: The developer inserts an additional data processing step before the state object is rendered in the UI, so that Wildcard can apply sorting and filtering. *Annotations*: The developer adds additional React components managed by Wildcard into their UI tree, which render additional data as annotations.

In certain types of software, customizability is often touted as a key selling point. An integrated website adapter would provide a way for developers to integrate with an existing ecosystem of formulas and customization tools, without needing to build all that functionality from scratch.

[1]So far we have only implemented AJAX scraping in the Firefox version of Wildcard, since Firefox has convenient APIs for intercepting requests. It appears possible to implement in Chrome as well, but we have not finished our implementation.

Shared storage adapter: It would be useful to share user annotations among users and across devices—for example, collaboratively taking notes with friends on 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, or through P2P connections that might provide stronger privacy guarantees.

3.2 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.

Queries are processed in three steps. First, the query invokes a primary DOM scraping table adapter that 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. Finally, the result table can be sorted and filtered by any column.

This query model can be viewed as a tiny (and rather constrained) subset of the SQL query model. Despite its simplicity, 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 paradigm of relational queries, it could theoretically be extended to support a wider range of queries.

The query engine is also responsible for executing formulas. We have built a small formula language resembling the ones used in visual query tools like SIEUFERD. As in those tools, and unlike in spreadsheets, formulas automatically apply across an entire column of data, and reference other column names instead of values on specific rows. For example, the user would write `=price * 2` to create a column with a value derived from a price attribute. This is more convenient than needing to copy-paste a formula across an entire column as in spreadsheets, and none of our customization use cases have uncovered a need for writing different formulas on different rows of the table.

3.3 Table editor

In Wildcard we provide a basic table editor as the user interface on top of the query engine. It is built with the Handsontable Javascript library, which provides UI elements for viewing and editing a table, as well as basic query operations like sorting and filtering.

todo: other table instruments

4 Key themes

4.1 Customization by direct manipulation

Hutchins, Hollan and Norman [7] describe a direct manipulation interface as one that uses a model-world metaphor, rather than a conversation metaphor. Rather than conversing with the system about an assumed world, "the world is explicitly represented" and the user can "[act] upon the objects of the task domain themselves."

Most GUIs today employ direct manipulation, but software customization tools typically use an imperative programming model, which implements the conversational metaphor rather than direct manipulation. For example, in Applescript [5], the scripting language for customizing Mac OS applications, here is how a user retrieves a list of of calendar names from the Calendar application:

```
tell application "Calendar"
  name of calendars
end tell
```

Some customization environments (Mac Automator, Zapier) forego text syntax and enable the user to connect programs and construct automations by dragging and dropping commands. These environments still do not constitute direct manipulation, though: the objects being manipulated are in the domain of programming, not the domain of the task at hand.

Imperative programming is a reasonable choice as the model for building customizations. Turing-complete programming provides a high ceiling for possible customizations, and a sequence of commands is a natural fit for automations which simulate a series of steps taken by the user.

However, there is a serious drawback to this approach. MacLean et al [10] describe an ideal for user-tailorable systems: a "gentle slope" from using to customizing, where small incremental increases in skill lead to corresponding increments of customization power. Requiring users to learn programming for customization creates an abrupt "cliff," where a significant learning investment is required even to implement simple customizations. Another goal of MacLean et al is that "it should be as easy to change the environment as it is to use it," at least for some subset of changes. In scripting languages, even user-friendly ones like Applescript or Chickenfoot, the experience of customization does not remotely resemble the experience of use, so these systems can't meet this goal.

With table-driven customization we aim to provide a gentler slope, by using direct manipulation for software customization. The data shown in the table view is the domain data from the original application. The user makes changes to the data by selecting areas of interest in the table, e.g. sorting/filtering by clicking the relevant column header, or adding annotation by clicking on the relevant row. These interactions are common in GUI applications, and Wildcard therefore meets MacLean et al's goal: some one-click customizations are as easy as using the original application.

One aspect of directness which we have chosen not to maintain in Wildcard is enabling customization in the context of the original user interface, as explored by some other tools like Scotty [6]. We have found that augmenting the original UI with an additional structured representation has numerous benefits. It provides a consistent experience across all applications, and clearly shows the user what structured data is available to work with.

Ainsworth et al provide a helpful taxonomy of the value of multiple representations [2]. In their terms, Wildcard plays a *complementary role* by supporting a different set of tasks from the original application, while displaying *shared information*. We also suspect that Wildcard helps *constructs deeper understanding* by *subtraction*. By stripping away details and only showing the essential data in an interface, Wildcard helps users think of new ways of using that data, outside the specific constraints of the original application.

It is important to note that there are many specific customizations that can be achieved in systems like Chickenfoot [4] that cannot be reproduced in Wildcard. We consider this an acceptable tradeoff in exchange for achieving a gentler slope of customization, and we show in Section 2 and Section 5 that our model can still implement many useful customizations in practice. Also, there is sometimes a way to reframe an imperative script in terms of our direct manipulation model; for example, a script that iterates through rows in a page adding some additional information could be reproduced using a formula in Wildcard. (*todo: mention the fact that the data abstraction step limits possibilities*)

4.2 Wrapping applications for customization

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

Ad hoc 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, ad hoc 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.

Anticipated customization tools, in contrast, 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 those that the official plugin API supports.

With Wildcard, we use a hybrid approach that aims to provide the best of both worlds. Programmers implement an API wrapper that is internally implemented as an ad hoc customization, but externally provides a high-level interface to the application, abstracting away the details of the user interface. These wrappers are added to a shared repository, available to all users of the system. When an end user is using a site that already has an adapter, they get a semantic customization experience that avoids low-level details.

One way to view this approach is as introducing a new abstraction barrier into third-party extension. Typically, a third party 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) handling the actual logic of the specific customization. Even though the mapping logic is often more generic than the specific customization, 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 repository of shared wrappers maintained by programmers, and a separate repository of specific customizations built on top of these wrappers. This general architecture has been successfully demonstrated 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.

(*todo: mention automated wrapper induction here or somewhere else?*)

5 Evaluation

We used Wildcard to customize 11 websites, including transactional sites like Amazon and Uber Eats, media consumption sites like Hacker News and Youtube, and a medical results entry platform managed by a member of our research group. (*todo: summarize all of these in a table*) So far, most usage has come from members of the project team, although we have had a few external users try the system.

Here we offer our reflections on using the system, focused on two key questions:

- What is the range of useful customizations possible in this paradigm?
- How feasible is it to build DOM scraping adapters for real websites, in practice?

5.1 Range of customizations

5.2 Viability of DOM scraping

In order for third-party customization through Wildcard to succeed, it is important that creating usable adapters for existing websites takes minimal effort.

Most of our adapters were created by members of our team. However, an external developer unaffiliated with the project contributed one adapter, designed to sort the Github page listing a user's repositories. They described the experience as "very straightforward."

```
'use strict';
import { urlMatches, extractNumber } from "../utils";
import { createDomScrapingAdapter } from "./domScrapingBase

const GithubAdapter = createDomScrapingAdapter({
  name: "Github",
  enabled: () => urlMatches(/github.com.*tab=repositories/)
  attributes: [
    { name: "name"    , type: "text"},
    { name: "stars"   , type: "numeric"},
    { name: "forks"   , type: "text"},
    { name: "updated", type: "text"},
    // TODO datetime type would be nice? not everything has
  ],
  scrapePage() {
    return Array.from(document.querySelectorAll("li.source"
      let name_el = el.querySelector('a[itemprop="name code
      let name = name_el.textContent.trim()

      let stars_el = el.querySelector('*[href*="/stargazers
      let stars = extractNumber(stars_el, 0)

      let forks_el = el.querySelector('*[href*="/network/me
      let forks = extractNumber(forks_el, 0)

      let lang_el = el.querySelector('*[itemprop="programmi
      // some repos don't have language set
      let lang = lang_el == null ? null : lang_el.textConte

      let updated_el = el.querySelector('relative-time')
      let updated = updated_el.getAttribute('datetime')

      return {
          id: name,
          rowElements: [el],
          dataValues: {
              name: name,
              stars: stars,
              forks: forks,
              updated: updated,
          },
      }
    })
  },
  onRowSelected: (row) => {
      row.rowElements.forEach(el => {
          if (el.style) {
              el.style["background-color"] = "#def3ff
```

```
            }
        });
        row.rowElements[0].scrollIntoView({ behavior: "smooth", block: "center" })
    },
    onRowUnselected: (row) => {
        row.rowElements.forEach(el => {
            if(el.style) {
                el.style["background-color"] = ``
            }
        })
    },
})

export default GithubAdapter;
```

The adapters for our test sites ranged from 36 to 117 lines of code, averaging 68 lines.

Some of the challenges of writing a DOM scraping adapter are the same ones involved with writing normal web scraping code, but the more interactive nature of Wildcard introduces additional challenges. One challenge is triggering updates to the spreadsheet data in response to UI changes that happen after initial page load. Site adapters are responsible for recognizing these changes by observing the DOM. So far, we have been able to use event listeners and the MutationObserver API to successfully observe changes, but it may prove challenging to observe changes on some sites only through the DOM. Another challenge is persisting updates to the DOM—some websites use virtual DOM frameworks that can occasionally overwrite changes made by Wildcard. Despite these challenges, we've managed to create an implementation that works around these issues in practice, and results in a usable customization experience for all the websites we've tried so far.

6 Related Work

This paper extends a workshop paper by Litt and Jackson [8] which presented an earlier prototype version of the Wildcard system. We build on their work in this paper by rearchitecting the internal implementation of the system around the table adapter abstraction, evaluating it on many more websites, and by more fully characterizing the behavior and design of the system.

6.1 Customization tools

- Buttons [10]: the original DM customization tool
- ScrAPIr: direct manipulation for APIs!
- customization tools
  - browser extensions
  - Thresher
  - Sifter
  - 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
  - visual customization tools:
    - Mac Automator
    - Zapier
    - IFTTT
    - These are weakly direct manipulation. But usually the thing you directly manipulate in these UIs is the *script*: not the actual *data* you want to operate on. VERY DIFFERENT.
  - desktop customization: Applescript, VBA, COM

6.2 Visual database query tools

- database GUIs:
  - Sieuferd [3]
  - Airtable [1]
  - other similar tools?
  - Liu & Jagadish: rules for a spreadsheet algebra for database queries [9]
  - Shneiderman paper
  - spreadsheets

6.3 Other

- web scraping / wrapper induction
- instrumental interaction, Scotty [6], Webstrates
- browser dev tools
- Self, Smalltalk
- multiple representations
- Semantic Web

References

[1] 2020. Airtable. https://airtable.com.

[2] Shaaron Ainsworth. 1999. The Functions of Multiple Representations. *Computers & Education* 33, 2-3 (Sept. 1999), 131–152. https://doi.org/10.1016/S0360-1315(99)00029-9

[3] Eirik Bakke and David R. Karger. 2016. Expressive Query Construction through Direct Manipulation of Nested Relational Results. In *Proceedings of the 2016 International Conference on Management of Data - SIGMOD '16*. ACM Press, San Francisco, California, USA, 1377–1392. https://doi.org/10.1145/2882903.2915210

[4] Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C. Miller. 2005. Automation and Customization of Rendered Web Pages. In *Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology - UIST '05*. ACM Press, Seattle, WA, USA, 163. https://doi.org/10.1145/1095034.1095062

[5] William R. Cook. 2007. AppleScript. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages - HOPL III*. ACM Press, San Diego, California, 1–1–1–21. https://doi.org/10.1145/1238844.1238845

[6] James R. Eagan, Michel Beaudouin-Lafon, and Wendy E. Mackay. 2011. Cracking the Cocoa Nut: User Interface Programming at Runtime. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology - UIST '11*. ACM Press, Santa Barbara, California, USA, 225. https://doi.org/10.1145/2047196.2047226

[7] Edwin L Hutchins, James D Hollan, and Donald A Norman. 1985. Direct Manipulation Interfaces. (1985), 28.

[8] Geoffrey Litt and Daniel Jackson. 2020. Wildcard: Spreadsheet-Driven Customization of Web Applications. In *Companion Proceedings of the 4th In- Ternational Conference on the Art, Science, and Engineering of Programming*. Association for Computing Machinery, Porto, Portugal., 10. https://doi.org/10.1145/3397537.3397541

[9] Bin Liu and H. V. Jagadish. 2009. A Spreadsheet Algebra for a Direct Data Manipulation Query Interface. In *2009 IEEE 25th International Conference on Data Engineering*. 417–428. https://doi.org/10.1109/ICDE.2009.34

[10] Allan MacLean, Kathleen Carter, Lennart Lövstrand, and Thomas Moran. 1990. User-Tailorable Systems: Pressing the Issues with Buttons. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems Empowering People - CHI '90*. ACM Press, Seattle, Washington, United States, 175–182. https://doi.org/10.1145/97243.97271

[11] B. Shneiderman. 1983. Direct Manipulation: A Step Beyond Programming Languages. *Computer* 16, 8 (Aug. 1983), 57–69. https://doi.org/10.1109/MC.1983.1654471