Creating a Cliché Library for Social Applications

by

Maryam Archie

S.B., Computer Science and Engineering, MIT, 2018

Submitted to the

Department of Electrical Engineering and Computer Science

in Partial Fulfilment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

June 2019

Author: ____

Department of Electrical Engineering and Computer Science
May 24, 2019

Certified by: ____

Daniel Jackson, Professor, Thesis Supervisor
May 24, 2019

Accepted by: ____

Katrina LaCurts, Chair, Master of Engineering Thesis Committee

Creating a Cliché Library for Social Applications

by

Maryam Archie

Submitted to the Department of Electrical Engineering and Computer Science on

May 24, 2019

in Partial Fulfilment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

Abstract

Déjà Vu is a platform for the end-user development of web applications. It reduces the complexity of web development by encapsulating common functionality across the web into a library of clichés. Users select clichés that represent core concepts in their application, specify various configuration variables and include actions that add the desired functionality to their application.

This thesis focuses on creating a cliché library for social applications (e.g. planning events, crowd-sourcing opinions, etc.). The catalogue of clichés and their actions were evaluated by replicating student projects from an undergraduate web development course. Most of the core functionality was replicated for each sample. The applications built with Déjà Vu will also serve as a reference and/ or starting point for future application developers.

Thesis Supervisor: Daniel Jackson

Title: Professor

Acknowledgements

I am very thankful for all of the advice, feedback and support from Santiago Perez De Rosso and Professor Daniel Jackson during my time spent on Déjà Vu. I would also like to thank my other collaborators, Ma Czarina Lao and Barry McNamara, for their useful feedback and coffee breaks together. Lastly, I would like to express my gratitude to my friends and family for their endless support on this journey.

Table of Contents

List of Figures

List of Tables

Chapter One: Introduction

Déjà Vu is a new programming paradigm that aims to make web development easier for non-developers. It exploits Jackson's conceptual theory [1] by identifying concepts commonly found in web applications (e.g. a comment on a post on Facebook[1] or a comment as a review for a product on Amazon[2]) and "translating" them into re-usable components known as *clichés.* Users can create complex web apps by choosing and combining cliché actions from a carefully curated cliché library in an HTML file.

Déjà Vu's cliché library provides users with the means to create complex web applications. For example, Déjà Vu simplifies the creation of applications for planning of social events and organization of groups by using at least the Event and Group clichés. These can be combined with any of the other clichés in the library to obtain the desired functionality. For example, an event can be bound to a location and thus, can be displayed on a map using the Geolocation cliché.

The research conducted for this thesis supplements the cliché library with new clichés and actions to support the development of social applications. This builds off the existing Déjà Vu platform, which abstracts the difficulties of web development

[1] Facebook. https://www.facebook.com/
[2] Amazon. https://www.amazon.com/

from the average user. At the inception of this project, the cliché library already contained various clichés and actions.

This work is important to the success of Déjà Vu because it makes possible apps that were impossible before. The updated cliché library will empower individuals (including novices) to create more complex applications that bring all desired functionality into one application as opposed to using multiple applications (e.g. email, polls and maps) to organize social events. Even though these augmentations were made to support specific scenarios in social applications, they can be used to create different types of applications. For example, the Schedule cliché could be used in a productivity application to plan meetings in the office, and the Passkey cliché could be used to grant students access to some restricted class material. In addition, the sample applications built as a part of this thesis serve as a guide or a stepping stone for building new applications with Déjà Vu, and their stylesheets could be reused in other applications.

1.1. Summary of Contributions

The following summarizes the contributions made to the Déjà Vu project:

- New clichés were added to the library to enable specific scenarios in some sample applications. For example, we had to create the Match cliché to replicate functionality in a sample application that required individuals to indicate interest in each other before being matched together.

- The cliché library was further augmented by complementing existing clichés with new actions. For example, to edit viewing permissions of a given resource, we added the `add-viewer`, `remove-viewer`, and `add-remove-viewer` actions to the Authorization cliché.
- We replicated the core functionality and styling of six sample social applications using the augmented cliché library. These included applications that allowed users to plan social gatherings and sporting events, invite people to events, aggregate opinions about places and topics and help individuals meet others with similar interests, availability and location. We used this experience to evaluate Déjà Vu and our new cliché extensions.

1.2. Thesis Overview

The rest of the thesis is structured as below:

- Chapter 2 gives a brief overview of the Déjà Vu platform and explains how an application, called SweetSpots, can be implemented with it.
- Chapter 3 discusses how clichés and actions are added to the library.
- Chapter 4 describes the other social applications that were built with Déjà Vu and highlights interesting use cases of clichés and actions in the applications.
- Chapter 5 reflects upon the project and the research experience
- Chapter 6 explores other bodies of work and explains what sets Déjà Vu apart from them.
- Chapter 7 summarizes the body of research presented in this thesis.

Chapter Two:
An Overview of Déjà Vu

The Déjà Vu platform consists of many parts to make it easier for individuals to create applications. Clichés and their actions are important for displaying data and interacting with it. They come together to form a cliché library, from which users can select clichés to add functionality to their applications. We describe this process with an example in Section 2.4.

2.1. What is a Cliché?

A cliché is a collection of actions with a common purpose. For example, the purpose of the Authentication cliché is to verify a user's identity with actions such as registering or logging in a user. Each action generally is accompanied by a user interface element that displays data to the user after fetching it from the server or accepts user input and sends it to the server for storage and processing. Thus, each cliché represents a vertical slice of the web application, abstracting implementation details from the user. A powerful feature of Déjà Vu is its compositional mechanism: actions can contain other actions and can be replaced by another action. This added flexibility allows users to customize and extend the functionality of their application. Creating a new cliché and adding actions to an existing one will be discussed in Chapter 3.

2.2. The Cliché Library

Together, these clichés form the cliché library which lists the purposes and actions of each cliché. With this library, users can quickly build applications because they do not have to worry about defining schemas, creating database tables and other server-side functionality. Table 1 shows the state of the library[3] as of May, 2019.

Cliché	**Purpose**
Allocator	Distribute resources among consumers
Authentication	Verify a user's identity with a username and password
Authorization	Control access to resources
Comment	Express an opinion or reaction in writing
Event	Schedule events
Follow	Receive updates from sources
Geolocation	Locate points of interest
Group	Manage users in aggregate
Label	Categorize items to be found later
Match	Connect users after attempting to match with each other
Passkey	Verify a user's identity with a code
Property	Describe an object with properties that have values
Ranking	Rank items
Rating	Crowdsource evaluation of items
Schedule	Find times to meet
Scoring	Keeps track of scores
Task	Keep track of pieces of work to be done
Transfer	Move resources between accounts

Table 1: A list of all the clichés and their purposes in Deja Vu's cliché library

[3] We expect the library to evolve as we develop more apps.

2.3. Building Applications

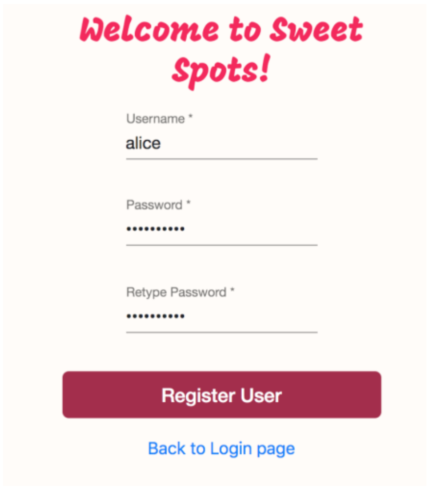
In Déjà Vu, all of these parts come together to help the user create applications. To build an application, the user first selects clichés from the cliché library and defines them in a configuration file. The user then creates app actions, either as pages or modular components, to structure the application. For example, a user can create an action that contains multiple cliché actions to modify data in an application. They can also create app actions to display application data using actions from various clichés. In addition, users can style their applications by writing custom CSS, using a Déjà Vu theme or using any popular CSS framework.

2.4. Example: SweetSpots

SweetSpots is a social application that allows users to map, filter and review points of interest on campus. Dubbed the "Yelp for Public Spaces", SweetSpots crowd-sources information about various locations and facilities around the MIT campus through posts, reviews, comments and voting. Users earn reputation for each Spot that they have posted, review that they have written for a Spot and vote that they have received for any of their reviews.

Let us walk through a scenario where a user, Alice, wants to submit the water fountain in McCormick Hall as a "sweet spot" on campus after which another user, Ben, favorites and reviews it:

a) Alice navigates to SweetSpots' registration page and makes an account with the username "alice".


Welcome to Sweet Spots!

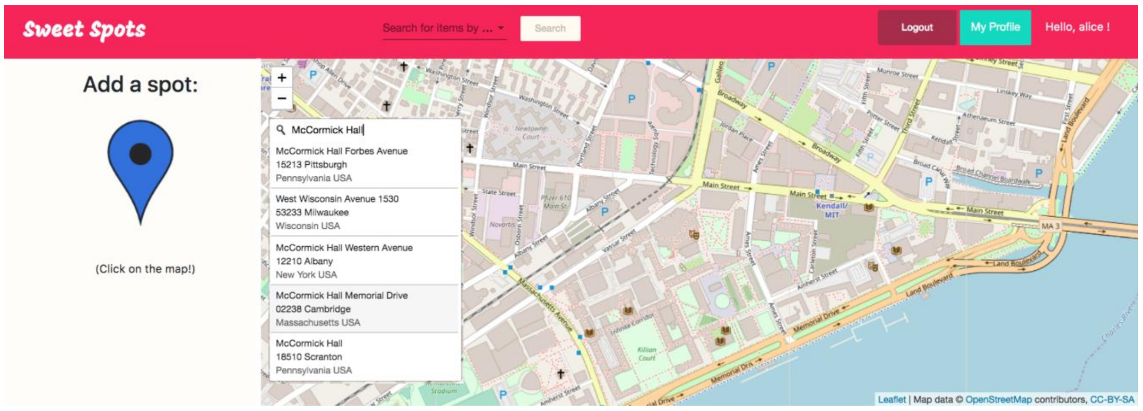
Username * alice

Password * ••••••••••

Retype Password * ••••••••••

Figure 1: The registration page for SweetSpots.

b) Alice is redirected to the main page with a map that displays crowd-sourced Spots, and searches for “McCormick Hall” in the map, opening the “Add a spot” form.



Add a spot:

(Click on the map!)

McCormick Hall

McCormick Hall Forbes Avenue
15213 Pittsburgh
Pennsylvania USA

West Wisconsin Avenue 1530
53233 Milwaukee
Wisconsin USA

McCormick Hall Western Avenue
12210 Albany
New York USA

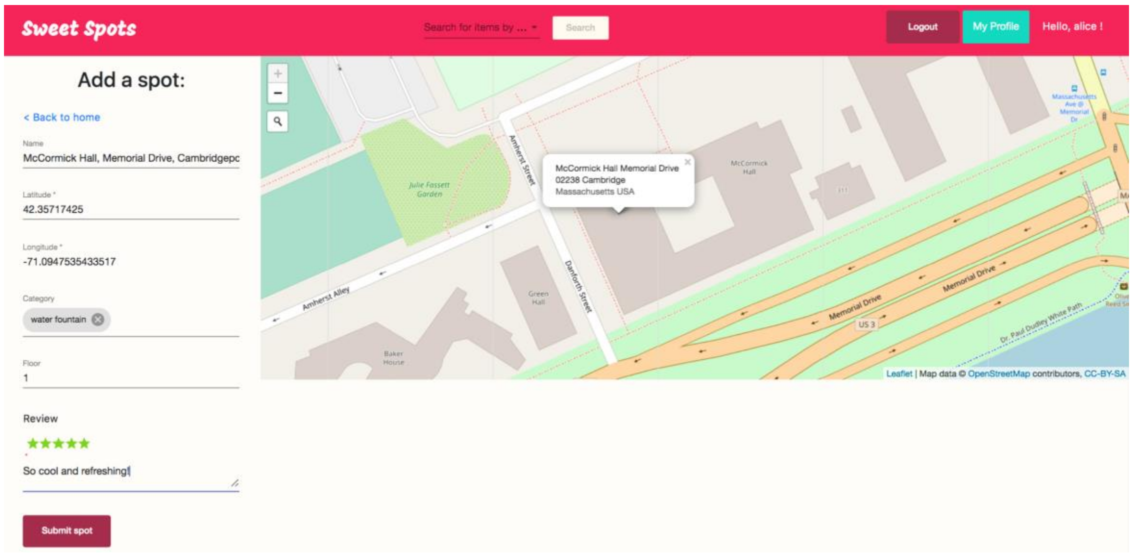
McCormick Hall Memorial Drive
02238 Cambridge
Massachusetts USA

McCormick Hall
18510 Scranton
Pennsylvania USA

Leaflet | Map data © OpenStreetMap contributors, CC-BY-SA

Figure 2: The main page for viewing and adding Spots in SweetSpots.

c) Alice fills out the rest of the information for the Spot and submits it.



Name
McCormick Hall, Memorial Drive, Cambridgepc

Latitude *
42.35717425

Longitude *
-71.0947535433517

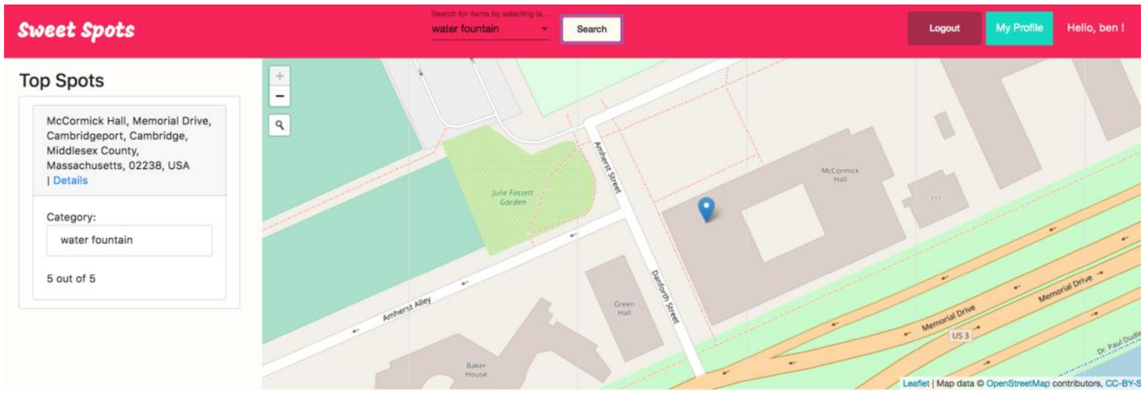
Category
water fountain

Floor
1

Review
★★★★★
So cool and refreshing!

Figure 3: The Add a Spot form in SweetSpots.

d) Ben logs in with his account and searches for water fountains on campus.


Sweet Spots
Search for items by selecting a...
water fountain
Search
Logout
My Profile
Hello, ben !
Leaflet | Map data © OpenStreetMap contributors, CC-BY-S

Top Spots

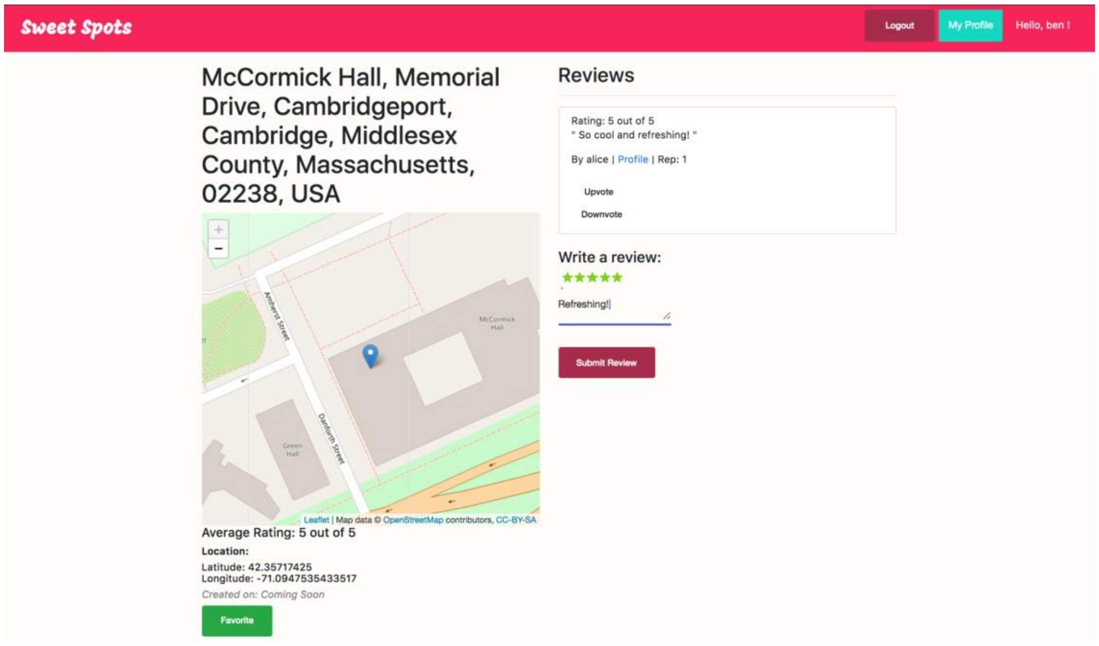
McCormick Hall, Memorial Drive, Cambridgeport, Cambridge, Middlesex County, Massachusetts, 02238, USA
| Details

Category:
water fountain

5 out of 5

Figure 4: Filtering Spots in SweetSpots.

e) After visiting the water fountain suggested by Alice, Ben upvotes her review of the Spot and writes his own review for the Spot. He also favorites the Alice's Spot for future reference.



McCormick Hall, Memorial Drive, Cambridgeport, Cambridge, Middlesex County, Massachusetts, 02238, USA

Reviews

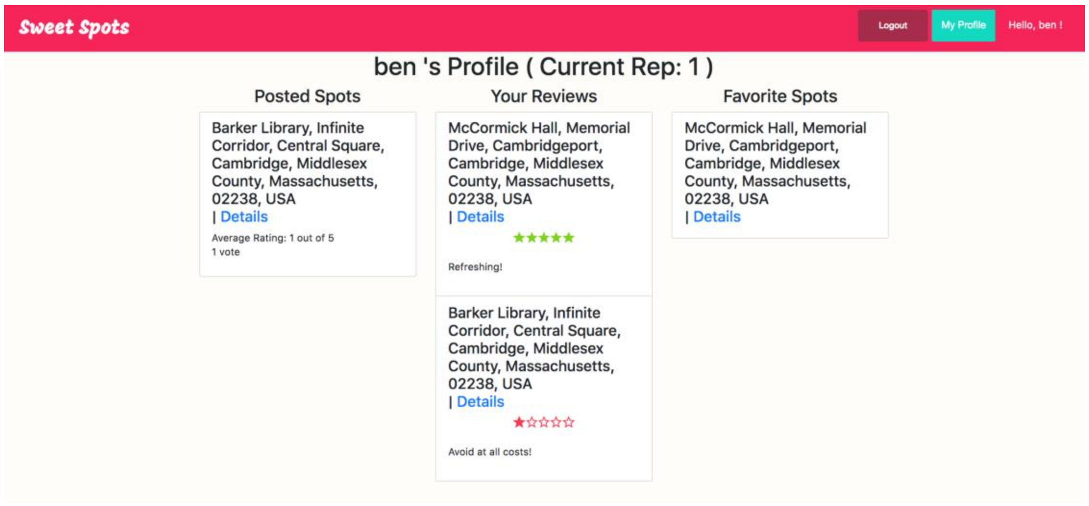
Rating: 5 out of 5
" So cool and refreshing! "
By alice | Profile | Rep: 1

Refreshing!

Average Rating: 5 out of 5
Location:
Latitude: 42.35717425
Longitude: -71.0947535433517
Created on: Coming Soon

Figure 5: The details page for a Spot in SweetSpots.

f) When Ben looks at his profile, he can see the Spots that he has already reviewed and favorited. After creating a Spot himself, his reputation score is 1.


Sweet Spots

Logout | My Profile | Hello, ben !

ben 's Profile (Current Rep: 1)

Posted Spots

Barker Library, Infinite Corridor, Central Square, Cambridge, Middlesex County, Massachusetts, 02238, USA
| Details

Average Rating: 1 out of 5
1 vote

Your Reviews

McCormick Hall, Memorial Drive, Cambridgeport, Cambridge, Middlesex County, Massachusetts, 02238, USA
| Details
★★★★★
Refreshing!

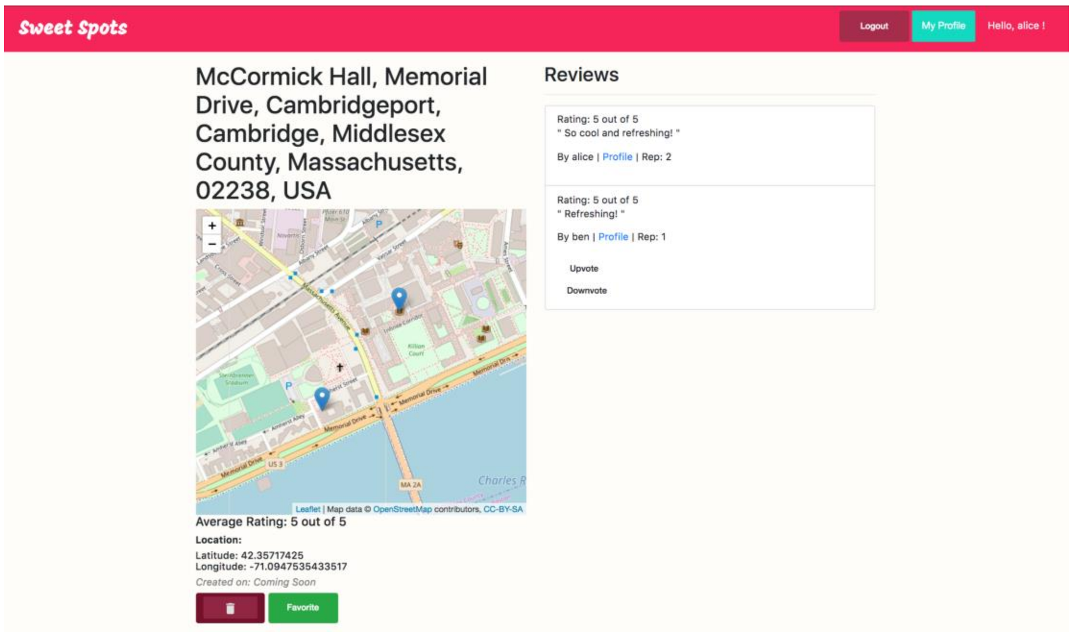
Barker Library, Infinite Corridor, Central Square, Cambridge, Middlesex County, Massachusetts, 02238, USA
| Details
★☆☆☆☆
Avoid at all costs!

Favorite Spots

McCormick Hall, Memorial Drive, Cambridgeport, Cambridge, Middlesex County, Massachusetts, 02238, USA
| Details

Figure 6: The user profile page in SweetSpots.

g) Looking back at her Spots, Alice sees that Ben has reviewed her water fountain spot. She can also see both of their reputation scores and vote on Ben's review, but not hers.


Sweet Spots

Logout | My Profile | Hello, alice !

McCormick Hall, Memorial Drive, Cambridgeport, Cambridge, Middlesex County, Massachusetts, 02238, USA

Leaflet | Map data © OpenStreetMap contributors, CC-BY-SA

Average Rating: 5 out of 5

Location:
Latitude: 42.35717425
Longitude: -71.0947535433517
Created on: Coming Soon

Favorite

Reviews

Rating: 5 out of 5
" So cool and refreshing! "
By alice | Profile | Rep: 2

Rating: 5 out of 5
" Refreshing! "
By ben | Profile | Rep: 1
Upvote
Downvote

Figure 7: The Spot author's view of the Spot details page.

2.4.1. Identifying Clichés to use in an Application

Conceptual design plays an integral role in developing applications with Déjà Vu. It encourages you to identify the core concepts of your application and how they interact with each other. These concepts can then be mapped to clichés to achieve the desired behaviors.

We identified several concepts in SweetSpots: Spots, Reputation, Categories, Reviews and Ratings. A Spot encapsulates information about a point of interest, including its name, location, and a classification (or Category). Each spot is reviewed at least once and each Review includes not only the author's opinion, but also a Rating. These Spots help people find nearby and greatly favored areas on the MIT campus. To reward users for submitting Spots and helpful Reviews, users are given Reputation.

In Table 2, we mapped these application concepts to ten instances of clichés and give examples of scenarios that they enable. The Authentication and Authorization clichés are used to handle user accounts and manage access to certain Spots. The Geolocation cliché is used to display Spots on the map and allow the user click on the map to autofill the coordinates and name of the Spot in the the "Add a spot" form. The Label cliché most aligns with the Category concept as they both classify items for faster lookup. We used the Comment cliché for Reviews because they are both used to express a user's opinion. The Rating cliché is used to rate a Spot and the Property cliché is used to record the floor information of a Spot. Since we want to allow users to bookmark (or favorite) Spots for future reference, we include the Follow cliché. Lastly, the Scoring cliché is used to keep track of a user's reputation and the scores of individual reviews.

Cliché	App Concept	Supported Scenarios
Authentication	User	Alice makes an account; Ben logs in to his account
Authorization	Spots	In the posted spots column of the profile page, Ben only sees Spots that he created
Geolocation	Spots (location)	Alice uses the map to identify a location to anchor the Spot; Ben sees previously created Spots on the map
Label	Category	Alice tags the Spot as a "water fountain"; Ben searches for Spots matching the "water fountain" category
Comment	Review	Alice gives the Spot a review saying that it is "So cool and refreshing!"; Ben also reviews the Spot saying that it is "Refreshing"
Rating	Rating	Alice gives the Spot a 5-star rating to reflect the Spot's high quality
Property	Spots (floor)	Alice indicates that the Spot is on the 1st floor of McCormick Hall
Follow	Spots (favorite)	Ben favorites the Alice's Spot; Ben sees his favorited spots on his profile page
Scoring	Reputation; Review (score)	Alice earns reputation for posting a Spot; Ben upvotes Alice's review, contributing the score of her review and her reputation.

Table 2: A list of the clichés used to replicate SweetSpots and the concepts and scenarios that they support.

Given its ability to store free-form text, we could have used the Property cliché for Reviews instead of the Comment cliché. However, the purpose of the Review is to share the user's opinion about a Spot, which aligns with the more specific objective of the Comment cliché (express an opinion or reaction in words) than that of the Property cliché (describe an object with properties that have values).

The Scoring cliché is actually used twice in the SweetSpots application to keep track of a review's score (sum of upvotes and downvotes) as well as a user's reputation (sum of number of posted spots and the sum of the scores of their reviews). We will discuss how to avoid name collisions and set configuration variables for other clichés (e.g. Geolocation and Property) in the next section.

2.4.2. Configuring the Application

Once the relevant clichés have been identified, we must configure the application to use them. This is done by defining a configuration file (`dvconfig.json`) and listing the clichés to be used in the application. In this file, we can also map application pages to different routes for easier navigation (Figure 8: lines 32 – 39).

Some clichés require extra configuration. For example, if we wanted to use the Leaflet Maps API with the Geolocation cliché in SweetSpots, we would have to set the "mapType" key in the config object of the Geolocation cliché to "leaflet". This is shown in Figure 8: lines 8 – 10. Since we are using the Property cliché to specify the floor location of a Spot, we need to define the schema as shown in Figure 8: lines 19 – 27. Lastly, we seed the Label database (Figure 8: lines 13 – 16) with categories such as "bathroom", "nap space" and "water fountain" to provide category suggestions when the user is creating a Spot.

```
{
  "name": "sweetspots",
  "usedCliches": {
    "authentication": {},
    "authorization": {},
    "comment": {},
    "rating": {},
    "geolocation": {
      "config": {"mapType": "leaflet" }
    },
    "follow": {},
    "label": {
      "config": {
        "initialLabelIds": ["bathroom", "nap space", "water fountain", "study
space"]
      }
    },
    "property": {
      "config": {
        "schema": {
          "title": "SpotDetails",
          "type": "object",
          "properties": {
            "floor": { "type": "string" }
          }
        }
      }
    },
    "reputation": { "name": "scoring"  },
    "reviewscore": { "name": "scoring" }
  },
  "routes": [
    { "path": "login", "action": "login" },
    { "path": "main",  "action": "main" },
    { "path": "profile", "action": "show-user-profile" },
    { "path": "register", "action": "register" },
    { "path": "spot-details", "action": "show-spot-details" },
    { "path": "", "action": "main" }
  ]
}
```

Figure 8: The configuration file (`dvconfig.json`) for SweetSpots. This file specifies which clichés will be used in the application and maps URL paths to page app actions for easier navigation.

Although we can have multiple instances of a single cliché in an application, it is important to alias them to avoid name collisions in the platform. Since we have two instances of the Scoring cliché (one to track review scores and the other to track a user's reputation), we alias one as "reputation" and "reviewscore" (Figure 8: lines 29 – 30). This not only prevents the collision problem, but also describes what they are used for.

2.4.3. Adding App Actions to the Application

Applications built with Déjà Vu are structured into pages, modular app components and cliché actions. Pages and app components are known as *app actions* and help organize the flow of the application. Déjà Vu is able to distinguish between cliché and app actions since the HTML content of app actions must begin and end with `dv.action` tags, and specify the action's name. The following discusses what is necessary to create the main view page of the SweetSpots application.

To create the `sweetspots.main` app action shown in Figure 9, we must first create a directory ("main") containing an HTML ("main.html") and CSS ("main.css") file. Before adding content to the HTML file, we indicate that `main` is an app action to Déjà Vu by enclosing the content of the HTML file with `<dv.action name="main"></dv.action>` tags. Now, we can include other app and cliché actions inside of the `sweetspots.main` action. Assuming we that we have already made a `navbar` action, we can include it in the main page as shown in Figure 9: line 2. The interactive map is also added to the page using the `display-map` action of the Geolocation cliché (Figure 9: Line 19); with one line of HTML, the user can view Spots on the map and also use the map to select locations when creating a Spot. In the original application, the "Add a spot" form would only be displayed to the user if they were signed in and the user clicked the map. We can support this behavior using `dv.if` conditional statements. To determine whether a user is signed in, we can check the output of the navbar like accessing fields in an object (`sweetspots.navbar.loggedInUserId`). If this value is undefined or null, i.e. no one is signed in, we will show a message asking the user to sign in. Otherwise, the form will appear.

```
<dv.action name="main">
  <sweetspots. navbar />

  <div id="left-main-div">
    <dv.if condition=sweetspots.navbar.loggedInUserId>
      <dv.if condition=geolocation.display-map.newMarker>
        <sweetspots.add-a-spot-form
          loggedInUserId=sweetspots.navbar.loggedInUserId
          marker=geolocation.display-map.newMarker/>
      </dv.if>
    </dv.if>

    <dv.if condition=!sweetspots.navbar.loggedInUserId>
        To use this feature, please sign in or register.
    </dv.if>
  </div>

  <div id="map-div">
    <geolocation.display-map id="mainMapId" showDirectionsControl=false />
  </div>
</dv.action>
```

Figure 9: The HTML code for the `main` app action in SweetSpots.

2.4.4. Creating a Resource (Spot)

One of the most common behaviors in applications is to create some resource. In SweetSpots, users create Spots, helping others find ideal locations that fit their needs. However, recall from Section 2.4.1 that creating a Spot requires actions from multiple clichés. Since these actions should be executed simultaneously, they are wrapped inside of `dv.tx` element tags. To ensure that the atoms created in the cliché collections all refer to the same Spot, we include a `dv.gen-id` action tag inside of the `dv.tx` tags and use its output (`dv.gen-id.id`) as ID inputs for the cliché actions. For example, the `label.attach-labels` action uses the `dv.gen-id.id` value as the `itemId` input (Figure 10: line 14) since Spots are Items in the data model of the Label cliché. The `targetId` input of the `rating.rate-target` action uses the value from `dv.gen-id.id` because the user (source) is rating the spot (target). All functionality necessary for creating a Spot is encapsulated in the `add-a-spot-form` app action as shown in Figure 10, and is used in the `main` app action (Figure 9: lines 7 – 9).

When Alice or Ben looks at the Add a Spot form on the main page (as in Figure 2), they see seven fields corresponding to actions from five clichés – `geolocation.create-marker`, `label.attach-labels`, `property.create-object`, `rating.rate-target` and `comment.create-comment` (Figure 10: lines 7 – 24). The `create-marker` action contains three input fields while the other four cliché actions each have one. To ensure that the creation of a Spot contributes to the user's reputation, we also include the `reputation.create-score` action. Since this is a side effect of creating a Spot, it is hidden from the user using the `hidden=true` attribute. Similarly, the `follow.create-publisher`, `authorization.create-resource` and `authentication.authenticate` actions are necessary for favoriting spots, allowing users to see only spots that they created and verifying that the logged-in user is the one that sends the request to the server, but do not need to visible to the user.

```
<dv.action name="add-a-spot-form">
  <h2 id="add-a-spot">Add a spot:</h2>
  <dv.link href="/">&lt; Back to home</dv.link>
  <dv.tx>
    <dv.gen-id />
    <dv.status savedText="New spot saved"/>
    <geolocation.create-marker id=dv.gen-id.id
      title=$marker?.title
      latitude=$marker?.latitude
      longitude=$marker?.longitude
      mapId="mainMapId"
      titleLabel="Name"
      showOptionToSubmit=false />
    <label.attach-labels itemId=dv.gen-id.id
      inputLabel="Category"
      showOptionToSubmit=false />
    <property.create-object id=dv.gen-id.id
      showOptionToSubmit=false />
    <p>Review</p>
    <rating.rate-target sourceId=$loggedInUserId targetId=dv.gen-id.id
      execOnClick=false />
    <comment.create-comment authorId=$loggedInUserId targetId=dv.gen-id.id
      inputLabel=""
      showOptionToSubmit=false />
    <reputation.create-score sourceId=$loggedInUserId
      targetId=$loggedInUserId
      value=1
      hidden=true />
    <follow.create-publisher id=dv.gen-id.id
      showOptionToSubmit=false hidden=true />
    <authorization.create-resource id=dv.gen-id.id
      ownerId=$loggedInUserId hidden=true />
    <authentication.authenticate id=$loggedInUserId hidden=true />
    <dv.button class="btn btn-primary">Submit spot</dv.button>
    <dv.callback defaultHref="/profile" onExecSuccess=true hidden=true />
  </dv.tx>
</dv.action>
```

Figure 10: The HTML code necessary for the add-a-spot-form action. The atoms created by each cliché action in the transaction are “bound” to each other by the ID generated by the dv.gen-id action.

We can also modify the default presentation of the cliché actions. For example, the original SweetSpots application did not have any placeholders in the review input element, but the default placeholder of the input field of the `create-comment` action is “Write your comment”. To replace the placeholder, we put an empty string as the value to the `inputLabel` input as shown in Figure 10: line 23. Similarly, most cliché actions that modify state have a submit button, but when combined with other cliché actions, we want to hide the individual submit buttons. This is done by setting the `showOptionToSubmit` input to false and including a `dv.button` element to execute all of the actions in between the `dv.tx` tags (e.g. Figure 10: line 13).

Upon a success, a confirmation message can be shown to the user before being redirected to another page. In SweetSpots, the user sees the "New spot created!" message after a spot was successfully created because the `dv.status` tag is also included inside of the `dv.tx` tags (Figure 10: line 6). To redirect a user to another page upon success, the `dv.callback` action is nested inside of the `dv.tx` tags, specifying the new location with the `defaultHref` input as shown in Figure 10: line 35.

2.4.5. Displaying the Resources (Spots)

Once we have created a resource, we should be able to view them using one or more `show-` cliché actions. In Figure 6, a user could see a list of spots that they created, reviewed and favorited on their profile page. This section explains how we enable the Spot viewing functionality in SweetSpots; in particular, we show how to only display Spots authored by the logged in user.

We could use a `show-` action from any one of the eight clichés used to in the creation of a Spot. However, these `show-` actions will list all Spots created, not just the ones created by the logged in user. To filter the Spots by author, we can use the `show-`

`resources` action of the Authorization cliché, ensuring to set the `createdBy` input as the ID of the logged in user. At this point, the user will see a list of Spot IDs and author IDs. This is because the `show-resources` action uses the `show-resource` action by default which only displays information stored in Authorization's resource collection. However, we want to show the name of the Spot, a link to its details page, its average rating and the number of times it was rated and reviewed.

```
1  <dv.action name="show-user-profile-details">
2    <div class="container">
3      ...
4  
5      <h3>Posted Spots</h3>
6      <authorization.show-resources createdBy=$loggedInUserId
7        noResourcesText="You haven't created any spots yet."
8        showResource=<sweetspots.show-spot id=$id /> />
9  
10     ...
11   </div>
12 </dv.action>
13 
```

Figure 11: Users can see spots that they created using the authorization.show-resources cliché action and replacing the default show-resource action to show more information about the Spots.

To do this, we replace the default `show-resource` action that the show-resources action uses to render the individual elements with an app action called `show-spot`. This is shown in Figure 11: line 8, where the `show-spot` action will use the ID that the `show-resources` action initially passes to the `show-resource` action. Using this ID (signified as an input by `$id`), the cliché actions inside of the `show-spot` action can load the relevant information. For example, the `geolocation.show-marker` action uses the `$id` input to load the name of the Spot (Figure 12: line 4) and the `rating.show-average-rating` cliché action uses it to load the average rating of the target that is associated with the Spot (Figure 12: lines 14 and 18).

```
1  <dv.action name="show-spot">
2    <h4>
3      <geolocation.show-marker class="inline-block"
4        id=$id
5        showId=false
6        showLatLong=false
7        showMapId=false /> |
8      <dv.link href="/spot-details" params={ id: $id } class="inline-block">
9        Details
10     </dv.link>
11   </h4>
12   <p>Average Rating:
13     <rating.show-average-rating class="inline-block"
14       targetId=$id
15       showStars=false
16       showNumRatings=false />
17     <br />
18     <rating.show-average-rating targetId=$id showValue=false />
19   </p>
20 </dv.action>
```

Figure 12: The show-spot app action shows the name and average rating of the Spot, given an ID.

2.4.6. Styling the Application

Users can style applications built with Déjà Vu by writing their own CSS rules, using themes or using third party UI component libraries.

Writing Custom CSS

This method of styling applications is the most powerful because it can override previous rules from stylesheet imports. Custom CSS, for example, allows us to modify the colors, position and size of HTML elements in an application.

Employing Third-Party Frameworks

When we build our clichés, we sometimes use external libraries that require styling that they package and distribute as CSS files. Most of our cliché actions use Angular Material and Bootstrap components and thus, we require users to import these stylesheets in their application's main stylesheet (`styles.css`). Since we are using the Rating and Geolocation clichés in SweetSpots, we also need to import stylesheets from external libraries[4]. These imports are shown in Figure 13.

```
/* Default */
@import "~@angular/material/prebuilt-themes/indigo-pink.css";
@import "~bootstrap/dist/css/bootstrap.min.css";

/* Rating */
@import "~css-star-rating/dist/css/star-rating.min.css";

/* Geolocation */
@import "~leaflet/dist/leaflet.css";
@import "~leaflet-routing-machine/dist/leaflet-routing-machine.css";
@import "~leaflet-control-geocoder/dist/Control.Geocoder.css";
```

Figure 13: SweetSpots' main stylesheet includes imports from external UI frameworks.

Using Pre-Defined Themes

To help users style their applications and transfer the stylings from one app to another, we proposed developing a series of themes for common app components, e.g. navigation bars, login pages, register pages and profile pages. This involved creating style sheets with special identifiers for use in applications.

[4] Currently, users have to read the cliché documentation to figure out what stylesheets (if any) they need to import. In future iterations of Déjà Vu, we intend to make it easier to import necessary stylesheets by either packaging them with the cliché or creating a master spreadsheet with all styles necessary for the clichés to work.

We made a prototype for navigation bars which typically include the title of the app, section names, the name of the signed in user and a sign out button. To use the theme, the user needs to import it into the application's main stylesheet `(styles.css)` and use the structure shown in Figure 14 to render the SweetSpots navigation bar. The navigation bar's contents must be wrapped in a `<nav>` element with the navbar class. Each navigation item, e.g. links and buttons, in the navigation bar is defined with the `navbar-item` class. These `navbar-item`s must be wrapped inside of a `div` or `ul` element with the `navbar-items-list` class.

```
<dv.action name="navbar" loggedInUserId$=authentication.logged-in.user.id
  searchResults$=label.search-items-by-labels.searchResultItems>

  ...

  <nav class="navbar navbar-light">
    <div class="navbar-items">
      <div class="navbar-item navbar-header">
        <dv.link class="sweet-spots-heading" href="/main">
          Sweet Spots
        </dv.link>
      </div>

      <div class="navbar-item navbar-left search">
        <label.search-items-by-labels hidden=!$showSearch />
      </div>

      <dv.if condition=authentication.logged-in.user>
        <div class="navbar-item navbar-right">
          <div class="btn-toolbar toolbar">
            <dv.tx>
              <authentication.sign-out class="navbar-signout"
                buttonLabel="Logout"/>
              <dv.link href="/login" hidden=true />
            </dv.tx>
            &nbsp;
            <dv.link href="/profile"
              class="mat-button btn btn-info profile-btn">
              My Profile
            </dv.link>
          </div>
        </div>
        <div class="navbar-item navbar-right">
          <span class="navbar-hello">Hello,
            <authentication.show-user class="inline"
              user=authentication.logged-in.user
              showId=false />! &nbsp;
          </span>
        </div>
      </dv.if>

      ...
    </div>
  </nav>
</dv.action>
```

Figure 14: The HTML code used to render the navigation bar in SweetSpots. DV's navbar theme styling will be applied to it because it follows certain rules, e.g. every element inside of the `<nav>` tags must have the `navbar-item` class.

Chapter Three: Augmenting the Cliché Library

The cliché library is an integral part of Déjà Vu because it provides out-of-the-box functionality and ready-to-use components complete with a user interface wired up to a server for processing and fetching data. Since one of the main goals of this project was to support the development of social applications, it was very important to augment the existing cliché library with some new clichés and actions. These modifications are highlighted in Table 3. This chapter lists our contributions to the library and describes the process of adding clichés and actions to the library.

Cliché	**Actions**
Allocator	Create Allocation, Delete Resource, Edit Consumer, Show Consumer
Authentication	Authenticate, Change Password, Choose User, Logged In, Register User, Show User, Show Users, **Show User Count,** Sign In, Sign Out
Authorization	**Add/Remove Viewer**, **Add Viewer**, Can Edit, Can View, Create Resource, **Delete Resource**, **Remove Viewer**, Show Owner, Show Resource, Show Resources **(filter by creator)**, **Show Resource Count**
Comment	Create Comment, **Delete Comment**, Edit Comment, Show Comment **(given an object, an ID or target and author IDs)**, Show Comments, **Show Comment Count**
Event	Choose and Show Series, Create Event, Create Series, Create Weekly Series, **Delete Event**, Show Event, Show Events **(filter by date)**, **Show Event Count**
Follow	Create Message, Create Publisher, Edit Message, Follow/ Unfollow, Show Follower, Show Followers, **Show Follower Count**, Show Message, Show Messages, **Show Message Count**, Show Publisher, Show Publishers, **Show Publisher Count**

Geolocation	Create Marker**, Delete Marker**, Display Map **(now supports Leaflet + Google maps)**, **Get Current Location**, Show Marker **(by ID)**, Show Markers **(filter by radius)**, **Show Marker Count**
Group	Add to Group, **Choose Group**, Create Group, **Delete Group**, Input Member, Join/ Leave, Show Group, Show Groups **(filter by member)**, **Show Group Count**, Show Member, Show Members, **Show Member Count**, Stage
Label[5]	**Attach Labels, Create Label, Search Items by Labels, Show Item, Show Items, Show Item Count, Show Label, Show Labels, Show Label Count**
Match	**Attempt Match**, **Create Match**, **Delete Match**, **Show Attempt**, **Show Attempts**, **Show Match**, **Show Matches**, **Withdraw Attempt**
Passkey	**Create Passkey, Logged In, Show Passkey, Sign In, Sign Out, Validate**
Property	Choose Object, Create Object, Create Objects, Create Property, Object Autocomplete, Show Object, Show Objects, Show URL
Ranking	Create Ranking, Show Fractional Ranking, Show Ranking, Show Rankings, Show Target
Rating	**Delete Rating**, **Delete Ratings**, Rate Target, Show Average Rating, Show Rating, **Show Rating Count**, Show Ratings by Target
Schedule	**Create Schedule**, **Delete Schedule**, **Show All Availability**, **Show Next Availability**, **Show Schedule**, **Show Slot**, **Show Slots**, **Update Schedule**
Scoring	Create Score, **Delete Score**, **Delete Scores**, Show Score, Show Target **(of a score)**, Show Targets by Score **(of a source)**, **Update Score**
Task	Approve Task, Claim Task, Complete Task, Create Due Date, Create Task, **Create Task for Assignees**, **Input Assignee**, Show Assignee, Show Task, Show Tasks, **Show Task Count**, **Stage**, Update Task
Transfer	Add to Balance, Create Item Count, Create Transfer, Input Amount, Input Item Counts, Input Money, Show Amount, Show Balance, Show Item Count, Show Item Counts, Show Transfer, Show Transfers

Table 3: The cliché catalog contains 18 clichés, each with a set of related actions. The cliché and action names highlighted in bold were introduced during research for this thesis.

[5] Some code for the label cliché existed before my joining this project. However, I had to reimplement it and add new actions.

3.1. Identifying and Designing New Clichés

As applications were built with Déjà Vu, we sometimes discovered that there wasn't a cliché in the library that supported a scenario in an application. In SweetSpots, users should be able to add Category tags to Spots, but at the time of its development with Déjà Vu, there was no cliché that allowed users to do so. Since many applications use the concept of a label (e.g. labels in Gmail[6], product types in Amazon, hashtags in Twitter[7]), we considered adding it to the catalog. However, before implementing a new cliché, we must ensure it meets a set of standard cliché design criteria.

A cliché must be motivated by a purpose, unique to those that already exist in the catalog. Its data model must be simple, yet rich enough to build applications. Lastly, actions in a cliché collectively must help achieve a cliché's purpose.

Let us consider the conceptual design of the Label concept, shown in Figure 15, that we will use to create our cliché. The purpose of the Label concept is to categorize items for easier lookup. The data model (or structure) of the Label concepts consists of two collections: Labels and Items. A Label can be mapped to zero or more Items, while an Item must have at least one Label. The behavior of the Label concept describes actions that the user can take in relation to Labels and Items. An example of a behavior of the Label concept is `addLabelsToItem`. If the item hasn't

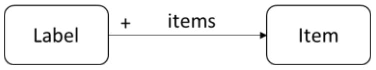
[6] Gmail. http://mail.google.com/mail/
[7] Twitter. https://twitter.com/

already been tagged by any of the input labels to the `addLabelsToItem` behavior, the item is added to the items collection of all labels that meet this criterion.

Title: Label

Purpose: Categorize items for easier lookup

Structure:



Behavior:

addLabelsToItem(labels: Label[], item: Item): Boolean

requires: item not in label.items

effects: ∀ l: Label in labels, l.items += item

removeLabelFromItem(label: Label, item: Item): Boolean

requires: item in label.items

effects: label.items -= item

deleteItem(item: Item): Boolean

requires: item in l.items for some l: Label

effects: ∀ l: Label, l.items -= item

searchItemsByLabels(labels: Label[]): Item[]

effects: { i: Item | label in i.labels }

Tactic:

If you add a label l to an item i, and later search for label l without removing the label from i in the meantime, then the result of the search will include i.

Figure 15: The conceptual design of the Label concept.

3.2. Implementing New Clichés

Once the design has been finalized, the cliché can be implemented. Using Déjà Vu's CLI, we can generate necessary configuration files, boilerplate code for the schema and server, and `create-`, `show-`, `update-` and `delete-` actions for the concept that the cliché represents. Typically, this is not enough to support all of the necessary actions for the cliché but it is a good starting ground. Next steps include:

a) Designing the schema

b) Implementing the server-side code

c) Designing the UI components

Figure 16 shows how these files interact together when a user a interacts with the UI component of a cliché action.

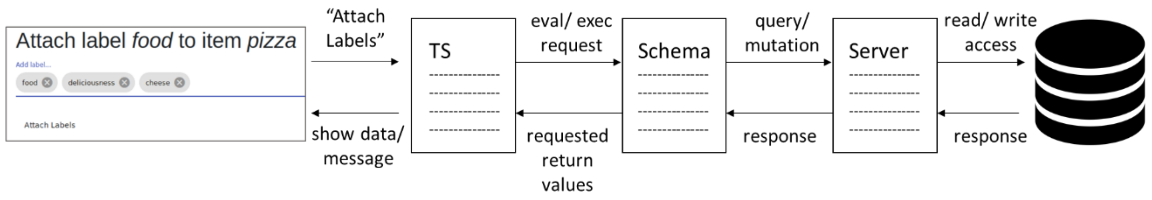


Figure 16: From left to right, when a user clicks the "Attach Labels" button in the rendered UI component, a request is sent to the cliché server and a query/mutation resolver is called depending on the GraphQL schema. Upon success, the database returns the value requested by the cliché action and updates the UI component accordingly.

3.2.1. Defining the schema

The first step in implementing a cliché is the design of the API that is used to send requests to and responses from cliché server and its action components respectively. The data model and specifications from the conceptual design of the cliché easily translates to a schema and queries and mutations with GraphQL[8]. Figure 17 shows the GraphQL schema for the Label cliché. To describe the data that can be fetched from the database, we define a Label object `type` like in lines 2 – 5 in Figure 17. This Label object `type` has two fields: a non-nullable ID and an array of IDs representing items that have that label.

The specifications translate to the Query and Mutation object types in the schema. To fetch a single label with scalar ID value, we include `label(id: ID!): Label` in the Query object `type`. For queries and mutations that require a more complex object as a parameter, we need to define `input` types, which are very similar to regular object types. For example, the input of the `addLabelsToItem` mutation is of the `AddLabelsToItemInput` type that must contain a non-nullable item ID and an array of non-nullable label IDs.

[8] GraphQL. https://graphql.org/

```
type Label {
  id: ID!
  itemIds: [ID]
}

# If no labels are provided, all items will be returned.
# Otherwise, only items with all specified labels will be returned.
input ItemsInput {
  labelIds: [ID]
}

# If no itemId provided, all labels will be returned.
# Otherwise, only labels of the given item will be returned.
input LabelsInput {
  itemId: ID
}

input AddLabelsToItemInput {
  itemId: ID!
  labelIds: [ID!]
}

input RemoveLabelFromItemInput {
  itemId: ID!
  labelId: ID!
}

type Query {
  label(id: ID!): Label
  labels(input: LabelsInput!): [Label]
  labelCount(input: LabelsInput!): Int
  items(input: ItemsInput!): [ID]
  itemCount(input: ItemsInput!): Int
}

type Mutation {
  createLabel(id: ID!): Label
  addLabelsToItem(input: AddLabelsToItemInput!): Boolean
  removeLabelFromItem(input: RemoveLabelFromItemInput!): Boolean
  deleteLabel(id: ID!): Boolean
}
```

Figure 17: The schema for the Label cliché. The Label type represent atoms in the Label database collection. The inputs of the query and mutation types are specified to indicate what the server is expecting.

3.2.2. Implementing the server-side code

The code that executes these queries and mutations with the MongoDB database lives in the server file. In this file, we can not only create indices for faster lookup, but also impose constraints (e.g. label IDs must be unique and labels can only have unique item IDs) in the non-relational database. The server-side code for the Label cliché can be found in Appendix A.1.

3.2.3. Designing the UI components

For each query and mutation in our GraphQL schema, there must be a client-facing UI component. Like Angular components, the code for the cliché actions are separated into HTML, TypeScript and CSS files.

HTML File

The HTML file contains the layout of the cliché action. Actions that mutate state are often presented as forms, while those that display state only show values that are not undefined. When designing actions like `show-labels`, it is important to use the child component (e.g. `show-label`) so that they can easily be replaced in applications if necessary.

CSS File

This is where cliché action-specific styling, e.g. setting the color of the success message to be green, is defined.

TypeScript File

The inputs and outputs of a cliché action are listed in its TypeScript file, in addition to the code responsible for sending requests to and receiving responses from the cliché server. Code that is run when a transaction is executed lives inside of the `dvOnExec()` function. Similarly, code that is run on evaluation is written in the `dvOnEval()` function.

The pre-generated `create-label`, `show-label` and `delete-label` actions were updated based on the desired behavior and new cliché actions, e.g. `add-labels-to-item` and `show-labels`, were generated using the CLI. The `update-label` action was removed from the cliché library because one should not be able to edit a label. If the Label object type contained both ID and name values, then it would make sense to have an `update-label` action.

Consider the `show-labels` action. In the TypeScript file, we define inputs for filtering the labels by item ID, modifying the message when there are no labels and replacing the default show-label action. Since we want to be able to use the show-label action in `show-labels`, we defined a public anchor variable `showLabels` that is necessary for the `dv-include` action in the HTML code. An output variable `loadedLabels` is also defined so that users can access this value and use it as input into another cliché or app action. Once the component loads and a connection is made with the cliché server, it executes the code inside of the `dvOnEval` block (Figure 26: lines 69 - 86). This sends a GET requires to the server with the inputs to the action and the desired format of the response. Once the labels are loaded, the component is re-rendered and the contents of the labels, i.e. IDs and item IDs, are displayed in the browser. The HTML and TypeScript code for `show-labels` can be found in Appendix A.2.

3.3. Adding Actions to Existing Clichés

Sometimes, we have all of the clichés necessary to build a certain application, but they are missing some actions needed to replicate all of the functionality of the application of interest. For example, to support the scenario in SweetSpots where Alice deletes the spot, we needed to add missing delete actions to clichés, e.g. `delete-marker` in the Geolocation cliché.

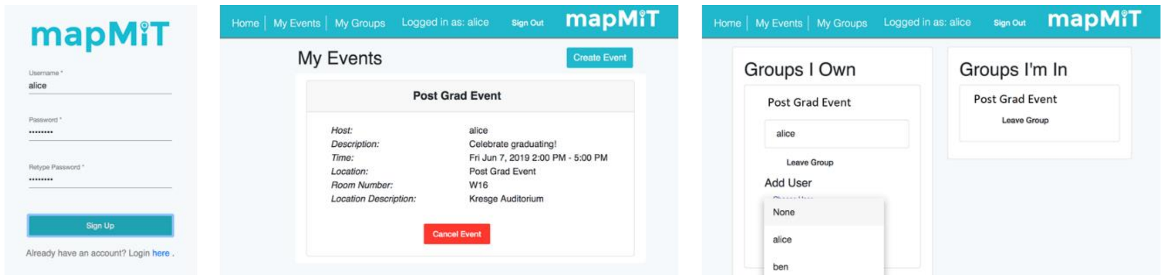
The process for adding an action to an existing cliché is similar to implementing cliché actions upon cliché creation. The cliché developer uses the CLI to generate the files for the new action. These files, in addition to the schema and server code, must be edited to deliver the desired behavior.

Chapter Four: Social Application Experiments

To evaluate the cliché library's ability to create social applications, we re-created six sample social-media applications, including SweetSpots. These applications were selected from the top twelve final projects of a popular undergraduate web development course at MIT. The teaching assistants, who are not associated with this project, identified these projects as the best in the class because of the clever manipulation of data, usage of APIs and UI/ UX. In this chapter, we will describe the sample social applications that were built with Déjà Vu and give examples that highlight the power of the Déjà Vu platform.

4.1. MapMIT

MapMIT, shown in Figure 18, displays on-campus events in a map, so that students can see publicized events. Users can create events and restrict visibility to certain groups of people. Users are also offered the option to join/ leave events. To replicate its core functionality, we used the Authentication, Authorization, Event, Geolocation, Group and Property clichés. Modelling an event as a group in the Group cliché allowed us to restrict visibility of events to groups of individuals. This was done by checking to see if the logged in user was a member of the group that was associated with the event.



Username * alice

My Events

Post Grad Event

Host:	alice
Description:	Celebrate graduating!
Time:	Fri Jun 7, 2019 2:00 PM - 5:00 PM
Location:	Post Grad Event
Room Number:	W16
Location Description:	Kresge Auditorium

Groups I Own

Post Grad Event

alice

Groups I'm In

Post Grad Event

Figure 18: The login, events and group pages in MapMIT.

4.2. Rendezvous

Rendezvous is an event coordinator and explorer for MIT students while they are off-campus. It helps MIT students foster connections with other students during their time away from MIT. Users can create events and see a list of nearby events in their feed or on the map as shown in Figure 19. Unlike other applications that used the Geolocation cliché, Rendezvous was the first to restrict the locations to a 50-mile radius of the user's current location. Thus, we added a new input "radius" to the `geolocation.display-map` action to filter markers within "radius" miles. If the "radius" input is left undefined, the `geolocation.display-map` action loads all of the markers for that map. As we continued to build applications, this modification proved beneficial. For example, Phoenix uses proximity as one of the criteria to suggest matches to users.

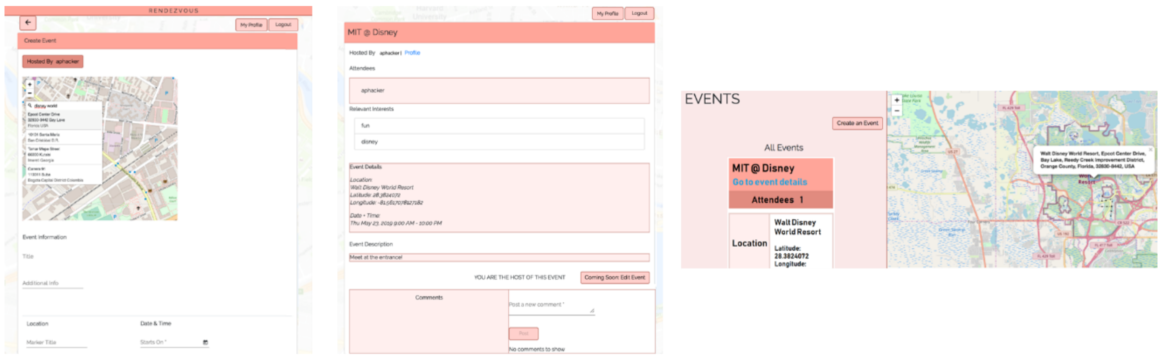
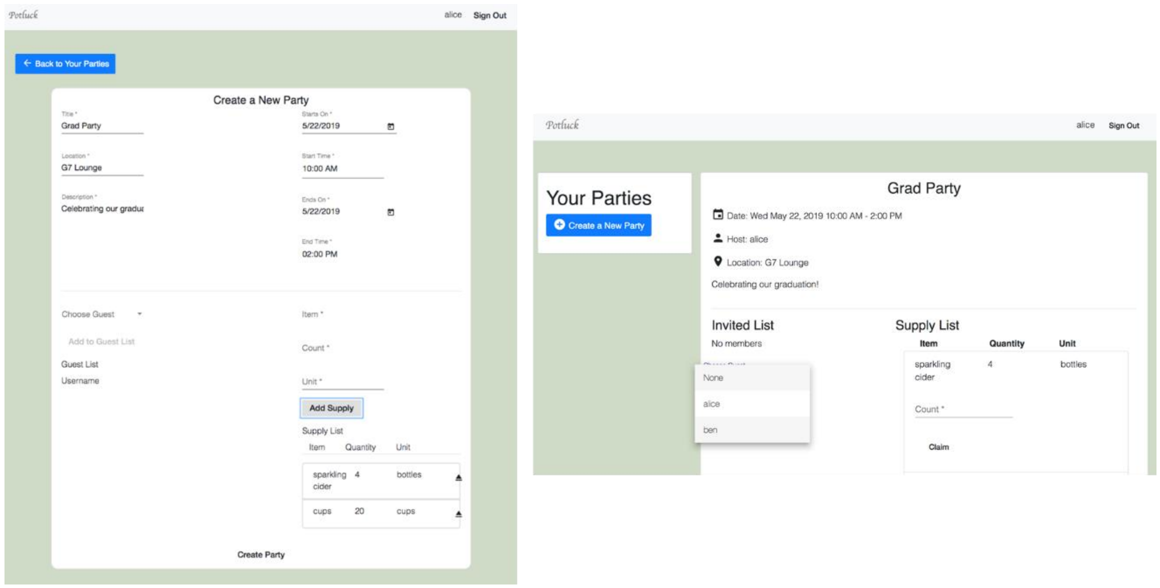


Figure 19: Creating and viewing an event in Rendezvous.

4.3. Potluck

Potluck is a party planning application to better organize party details, such as location, host information, guest lists and supplies. Hosts can list supplies necessary for the party and guests can indicate the quantity of items that they are willing to contribute. The Transfer cliché was used with a `dv.stage` action to support the scenario where users queue items that they would like to claim, as shown in Figure 20.



Create a New Party

Title *
Grad Party

Location *
G7 Lounge

Description *
Celebrating our gradua

Starts On *
5/22/2019

Start Time *
10:00 AM

Ends On *
5/22/2019

End Time *
02:00 PM

Guest List
Username

Supply List

Item	Quantity	Unit
sparkling cider	4	bottles
cups	20	cups

Grad Party

Date: Wed May 22, 2019 10:00 AM - 2:00 PM

Host: alice

Location: G7 Lounge

Celebrating our graduation!

Invited List
No members

Supply List

Item	Quantity	Unit
sparkling cider	4	bottles

Figure 20: Creating a new party with Potluck; Adding registered members to a party's guest list.

4.4. LiveScorecard

With LiveScorecard (Figure 21), individuals can host climbing competitions and allow participants to log their scores so that others can see the leader board in real-time. This application featured a unique authentication system, such that climbers required a competition code and climber code to sign in and log their scores, and spectators could only view the leader board if they had the code for the competition. This necessitated the creation of the Passkey cliché in which a user's passkey served both as an identifier and password.

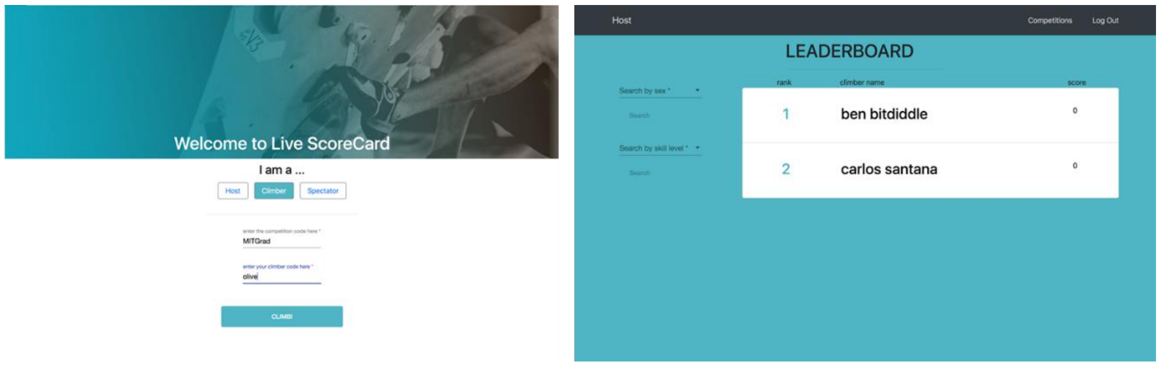
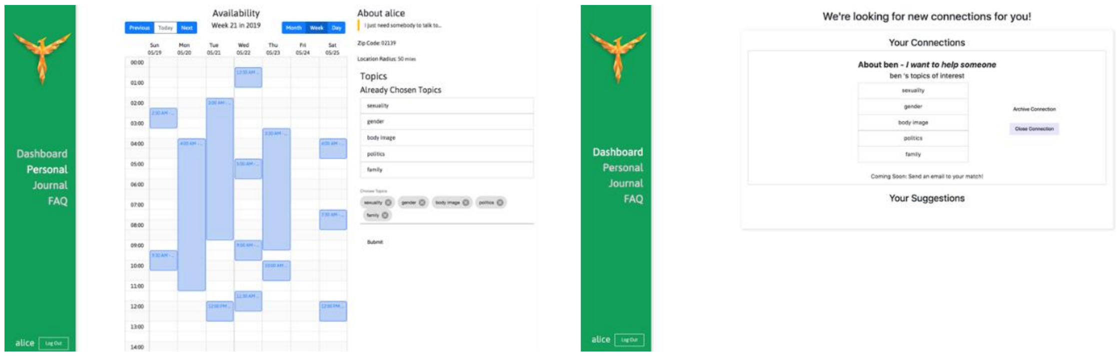


Figure 21: Climbers must login with both a competition and a climber code; Hosts, climbers and spectators can view the leaderboard of a competition.

4.5. Phoenix

Phoenix connects people afflicted with depression if they share similar interests, are located within each other's comfort range and have overlapping availabilities. However, users are only connected if they indicate interest in each other (as in dating apps like Tinder[9]). Thus, we had to create a Match cliché to support this scenario (Figure 22: right). Similarly, we had to create a Schedule cliché (Figure 22: left) so that users could indicate their availability and then determine if there were any overlaps amongst the availabilities of all registered users.

[9] Tinder. https://tinder.com/



About alice

I just need somebody to talk to...

Zip Code: 02139

Location Radius: 50 miles

Topics

Already Chosen Topics

sexuality
gender
body image
politics
family

Chosen Topics: sexuality, gender, body image, politics, family

Submit

We're looking for new connections for you!

Your Connections

About ben - *I want to help someone*

ben 's topics of interest

sexuality
gender
body image
politics
family

Archive Connection

Close Connection

Coming Soon: Send an email to your match!

Your Suggestions

Figure 22: A user's profile page including their personal information, topics of interest and availability for the week; Alice and Ben indicated interest in each other and are now connected.

4.6. SweetSpots

SweetSpots crowdsources and aggregates opinions about points of interest around the MIT campus. The SweetSpots application is discussed in greater detail in Section 2.4.

Chapter Five: Experience

This chapter summarizes our experience augmenting the cliché library and building applications using Déjà Vu.

5.1. Designing and Implementing Clichés

The following describes our experience designing and implementing new clichés for the catalog.

As described in Section 3.1, designing new clichés can be quite challenging. Interestingly, it took more time to design new clichés than to implement them. This often involved many conversations, whiteboard discussions and design revisions. Although it may have seemed overwhelming at times, we valued the importance of this process because we wanted to ensure that there was no overlapping functionality with an existing cliché.

In general, it is good practice to follow some standard when implementing user interfaces. However, as both a cliché and application developer, the importance of standardization became even more apparent. For example, when an individual implements a cliché, it is common to include `create-`, `update-`, `delete-` and `show-` actions. In addition, there are usually two `show-` actions – one to display a single item and another to display multiple. It has also become common practice to ensure that the action that displays multiple items uses the one that displays only one item. There is also a pattern for the inputs and outputs of the cliché actions. Inputs usually involve resource identifiers (e.g. `id`) and presentation modifiers (e.g. `showOptionToSubmit`), and outputs should be of the form "`loadedResource`". Thus, when the end-user builds application, they can rely on this standardization to build their applications quickly without constantly having to reference the documentation.

5.2. Reusing Clichés in Different Contexts

One of the most powerful features of Déjà Vu is the ability to use the same clichés to support different scenarios across multiple applications. Table 4 provides a breakdown of the clichés used in each sample social application built with Déjà Vu.

Over half of the social applications used the Event cliché to help schedule events, but the cliché wasn't explicitly mapped to an event concept in all applications. For example, LiveScorecard models an event in the Event cliché as a competition so that we can indicate start and end times for climbing competitions. In MapMIT, the Event cliché is used in the more traditional sense, i.e. it is used to plan events for social gatherings.

In four of the six applications, the Group cliché is used to organize members into groups so that they can be handled collectively. Potluck uses this cliché to support guest list functionality. LiveScorecard, on the other hand, uses three instances of the Group cliché to manage hosts, climbers and climbs separately in a competition.

The Label cliché was typically used in most applications to classify items for quicker lookup. In SweetSpots, Spots are given categories so that users can narrow their search for a “sweet spot”. The Label cliché is used in the Phoenix application so that users can indicate topics that they would like to discuss with others. Matches are suggested to users based on an overlap of users’ interests. Although the concepts in the applications, categories and topics, have different names, they aim to achieve a similar goal.

Our last example of cliché reuse involves the Scoring cliché in LiveScorecard and SweetSpots. It is used to keep track of the points logged by climbers in LiveScorecard. SweetSpots uses two instances of the Scoring cliché to keep track of the number of upvotes of a review and to keep track of a user’s reputation.

Some clichés however, e.g. Match and Schedule, are only used once. This is understandable given the different goals of the applications. It is also possible that the students envisioned additional functionality, e.g. matching based on location in Rendezvous, to their applications, but were constrained by the time allotted for the final projects.

	MapMIT	Rendezvous	LiveScorecard	Potluck	Phoenix	SweetSpots
Authentication	1	1	1	1	1	1
Authorization	1	1	1	1	1	1
Comment		1			1	1
Event	1	1	1	1		
Follow						1
Geolocation	1	1			1	1
Group	1	1	3	1		
Label		1	2		1	1
Match					1	
Passkey			2			
Property	1	2	3	2	2	1
Ranking						
Rating						1
Schedule					1	
Scoring			2			2
Task			1			
Transfer				1		
Total	6	8	7	5	8	8
Total Instances	6	9	13	6	9	8

Table 4: The clichés used to build the social applications.

5.3. Incrementing Functionality in Applications

Without Déjà Vu, adding new concepts to an application requires the developer to design a new schema and API, and create new front-end components. Sometimes, the developer has to modify the application's architecture to support some new functionality. With Déjà Vu, incrementing functionality of applications is as easy as adding the desired cliché concept to the application's `dvconfig.json` file.

For example, the Phoenix application was built in two stages – without and with the Schedule cliché. Initially, the Schedule cliché did not exist in the library, so we replicated parts of the application unrelated to scheduling. Once the Schedule cliché was designed and implemented, we added it to the Phoenix application by including it in the `usedCliches` object in the configuration file and added the necessary cliché actions to the relevant app actions.

Chapter Six: Related Work

Déjà Vu is not the only platform seeking to make it easier for individuals to design and build social web applications. Content Management Systems and research projects, such as Chorus [2] and the Ur/Web People Organizer [3], have all attempted to solve this problem with varying degrees of success (in terms of ease of use, richness of library components and adoption).

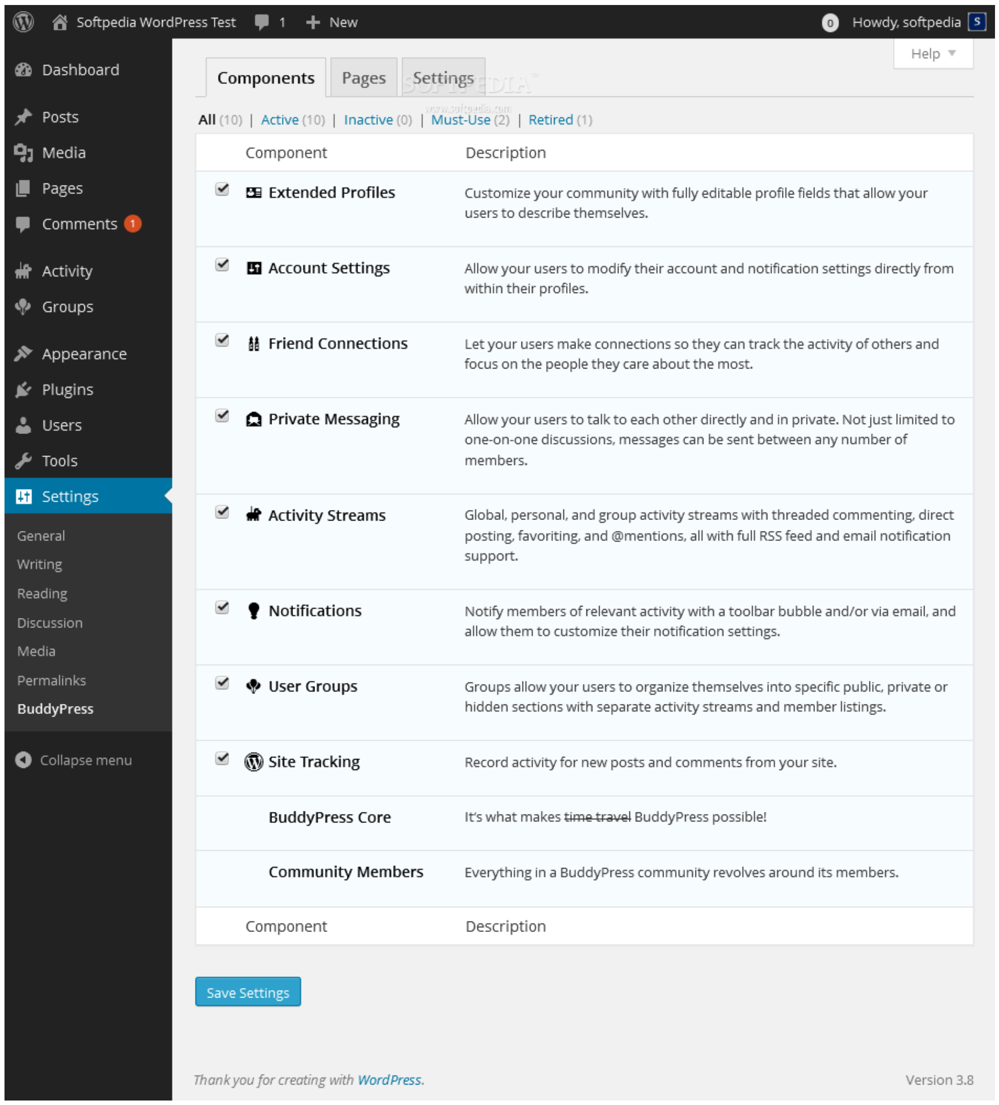
Content Management Systems, like WordPress[10] and Drupal Cloud[11], are widely used to publish content on the web due to the ease of use of WYSIWYG builders. Popular social applications built with WordPress, e.g. blogs by major tech companies such as Glassdoor[12], Yelp[13] and Facebook[14] [4], are built by experienced individuals, often using plugins such as Shortcake[15] and BuddyPress[16] [5]. Figure 23 shows an example of how common functionality in social applications are added to WordPress sites. Similarly, individuals and organizations have used Drupal Cloud to create social applications like Pinterest[17] [6] [7] and Global Dev Hub[18] [8]. Social applications built

[10] WordPress. https://wordpress.org/
[11] Drupal Cloud. https://www.drupal.org/
[12] Glassdoor. https://www.glassdoor.com/blog/
[13] Yelp. https://blog.yelp.com/
[14] Facebook. https://newsroom.fb.com/
[15] Shortcake (Shortcode UI). https://wordpress.org/plugins/shortcode-ui/
[16] BuddyPress. https://wordpress.org/plugins/buddypress/
[17] Pinterest. https://www.pinterest.com/
[18] GlobalDevHub. https://www.globaldevhub.org/

on top of Drupal tend to use similar modules, e.g. Open Social project[19], which provides features such as user profiles, groups, events, notifications and chat [9]. However, the problem of sharing information among plugins still arises, forcing users to write additional code, look for a plugin with the desired functionality or abandon the idea. Déjà Vu features a composition mechanism that sets it apart from Content Management Systems.

For example, in SweetSpots, users accrued "reputation" based on the number of posts and comments they made, as well as the number of times users favorited their post and number of votes they received on their comments. Since the posts and comments are separate entities, there is no easy way to keep track of these scores in WordPress or Drupal Cloud. This problem is easily solved by Déjà Vu. Clichés and actions in Déjà Vu can communicate with each other with its unique binding mechanism, allowing a single item to be mapped to multiple clichés. With Déjà Vu, the user can keep track of a user's "reputation" using the Property, Comment and Scoring clichés, such that when an update is made to a post or comment, the user's reputation is updated.

[19] Open Social. https://www.drupal.org/project/social



	Component	Description
✓	Extended Profiles	Customize your community with fully editable profile fields that allow your users to describe themselves.
✓	Account Settings	Allow your users to modify their account and notification settings directly from within their profiles.
✓	Friend Connections	Let your users make connections so they can track the activity of others and focus on the people they care about the most.
✓	Private Messaging	Allow your users to talk to each other directly and in private. Not just limited to one-on-one discussions, messages can be sent between any number of members.
✓	Activity Streams	Global, personal, and group activity streams with threaded commenting, direct posting, favoriting, and @mentions, all with full RSS feed and email notification support.
✓	Notifications	Notify members of relevant activity with a toolbar bubble and/or via email, and allow them to customize their notification settings.
✓	User Groups	Groups allow your users to organize themselves into specific public, private or hidden sections with separate activity streams and member listings.
✓	Site Tracking	Record activity for new posts and comments from your site.
	BuddyPress Core	It's what makes ~~time travel~~ BuddyPress possible!
	Community Members	Everything in a BuddyPress community revolves around its members.
	Component	Description

Figure 23: With BuddyPress, adding concepts like groups and chat are as easy as checking a box in WordPress' setting page. Image downloaded from https://scripts-cdn.softpedia.com/screenshots/BuddyPress_1.png in April 2019.

Like WordPress, the Chorus project[20] enables users to create applications with a WYSIWIG builder [2]. It also introduced the idea of social datatypes, i.e. components that can be put together to facilitate common conversational patterns. In the designer

[20] Chorus. http://www.chorus-home.org

mode, the user can select a combination of these social datatypes - lists, polls, forms, choices and/or primitive values (or atoms) - to create the desired interface for their audience. Although these social datatypes can be joined to form components in Chorus' Collaborative Document, Chorus still lacks a compositional mechanism for combining the functionality of individual datatypes into a more complicated one.

The Ur/Web programming model [10] was developed at MIT to facilitate the creation of modern, secure and dynamic web applications. It is a statically typed functional programming language that can be compiled to more popular web-specific languages supported by browsers. Using this model, the Ur/ Web People Organizer (UPO) platform [3] was created to simplify the creation of applications for organizing people and events, i.e. it is focused on a different niche (productivity) than the social niche explored in this thesis.

Although the user can select UI components from a library in UPO, they are responsible for specifying the data model and writing a combination of Ur (similar to ML and Haskell), XML and SQL-like database queries. These components either bind to the user-defined data model or maintain their own state. With Déjà Vu, these implementation details are abstracted away from the user. To build an application, the user only has to write HTML and CSS. In addition, the user does not have to worry about defining any data models for their application since state is maintained by the individual clichés themselves. Lastly, widgets in UPO are more aligned with basic input and output structures (e.g. input – generic user input, editGrid – an editable grid, row – a row in a table) and application-specific functionality (e.g. finalGrades – displays the finals grades of students in a class) than concepts motivated by a purpose.

Chapter Seven: Conclusion

This thesis provides an overview of the Déjà Vu platform and describes our contributions to its cliché library and evaluation. The research conducted as part of this thesis is important for making the catalog more robust, allowing people to create complex social applications much faster with Déjà Vu than from scratch. Using the SweetSpots example, we explained how applications are built with Déjà Vu and described the process of augmenting the cliché library. Upon reflection, there were many successes as well as areas of improvement for this project, and Déjà Vu as a whole.

The Déjà Vu platform significantly reduces the time to build an application from scratch. Each student sample project represented four 6-person weeks-worth of work while we re-created each within a week. We hope that our building a suite of sample applications will provide inspiration for users and help reduce the time taken to build apps of similar caliber even further.

The conceptual design of an application before implementation is an important part of application development. As a part of the student final projects, they had to identify and define the main concepts in their applications. Since they were already defined, there was a clear mapping of concepts to clichés in the catalog.

The cliché catalog is not yet complete. For example, we think that Search/ Filter, Log and Recommendation clichés would be helpful additions to the catalog. However, after briefly reviewing the final project presentations of the Fall 2018 iteration of the same undergraduate web development course, we believe that our catalog is robust enough to replicate the core functionality of at least the social applications that were presented.

Overall, I am grateful for the opportunity to collaborate on this project and firmly believe that it is a powerful tool that can help people create not only social applications, but also other types of applications. I am in awe of what we are able to achieve with Déjà Vu today and am excited to see how it evolves.

References

[1] D. Jackson, "A new modularity for software," November 2018. [Online]. Available: http://people.csail.mit.edu/dnj/talks/splash18/splash-talk-2018-no-builds.pdf. [Accessed 3 December 2018].

[2] J. L. Chen, Chorus: End User Programming of Social Applications, Cambridge: Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science., 2017.

[3] A. Chlipala, "The Ur/Web People Organizer," 2015. [Online]. Available: http://upo.csail.mit.edu/. [Accessed 6 December 2018].

[4] WPBeginner, "40+ Most Notable Big Name Brands that are Using WordPress," 8 March 2019. [Online]. Available: https://www.wpbeginner.com/showcase/40-most-notable-big-name-brands-that-are-using-wordpress/. [Accessed 19 May 2019].

[5] WordPress, "Case Studies - Enterprise WordPress hosting, support, and consulting - WordPress VIP," WordPress, 2018. [Online]. Available: https://vip.wordpress.com/case-studies. [Accessed 26 December 2018].

[6] Drupal, "Pinterest | Drupal.org," [Online]. Available: https://www.drupal.org/pinterest. [Accessed 20 May 2019].

[7] A. Stone, "Pinterest for Business | Drupal.org," 31 October 2017. [Online]. Available: https://www.drupal.org/case-study/pinterest-for-business. [Accessed 20 May 2019].

[8] GoalGorilla, "GlobalDevHub, an Open Social platform for the United Nations," 5 April 2017. [Online]. Available: https://www.drupal.org/case-study/globaldevhub-an-open-social-platform-for-the-united-nations. [Accessed 20 May 2019].

[9] GoalGorilla, "Open Social, a Community and Extranet Solution," Drupal, 23 June 2016. [Online]. Available: https://www.drupal.org/case-study/open-social-a-community-and-extranet-solution. [Accessed 16 December 2018].

[10] A. Chlipala, "Ur/Web: A Simple Model for Programming the Web," *Communications of the ACM,* vol. 59, no. 8, 2016.

Appendix A

A.1. The Server-Side Code for the Label Cliché

```
import {
  ActionRequestTable, ClicheDb, ClicheServer, ClicheServerBuilder,
  Collection, Config, Context, EMPTY_CONTEXT, getReturnFields
} from '@deja-vu/cliche-server';
import { IResolvers } from 'graphql-tools';
import * as _ from 'lodash';
import {
  AddLabelsToItemInput, ItemsInput, LabelDoc, LabelsInput
} from './schema';

import { v4 as uuid } from 'uuid';

interface LabelConfig extends Config {
  initialLabelIds: string[];
}

function standardizeLabel(id: string): string {
  return id.trim()
    .toLowerCase();
}

const actionRequestTable: ActionRequestTable = {
  'attach-labels': (extraInfo) => `
    mutation AttachLabels($input: AddLabelsToItemInput!) {
      addLabelsToItem(input: $input) ${getReturnFields(extraInfo)}
    }
  `,
  'create-label': (extraInfo) => `
    mutation CreateLabel($id: ID!) {
      createLabel(id: $id) ${getReturnFields(extraInfo)}
    }
  `,
  'search-items-by-labels': (extraInfo) => {
    switch (extraInfo.action) {
      case 'items':
        return `
          query SearchItemsByLabel($input: ItemsInput!) {
            items(input: $input) ${getReturnFields(extraInfo)}
          }
        `;
      case 'labels':
        return `
          query SearchItemsByLabel($input: LabelsInput!) {
            labels(input: $input) ${getReturnFields(extraInfo)}
          }
        `;
      default:
        throw new Error('Need to specify extraInfo.action');
    }
  },
  'show-items': (extraInfo) => `
    query ShowItems($input: ItemsInput!) {
      items(input: $input) ${getReturnFields(extraInfo)}
    }
  `,
  'show-item-count': (extraInfo) => `
    query ShowItemCount($input: ItemsInput!) {
      itemCount(input: $input) ${getReturnFields(extraInfo)}
    }
  `,
  'show-labels': (extraInfo) => `
    query ShowLabels($input: LabelsInput!) {
      labels(input: $input) ${getReturnFields(extraInfo)}
    }
  `,
  'show-label-count': (extraInfo) => `
    query ShowLabelCount($input: LabelsInput!) {
      labelCount(input: $input) ${getReturnFields(extraInfo)}
    }
  `
};

function getLabelFilter(input: LabelsInput) {
  const filter = { pending: { $exists: false } };
  if (!_.isNil(input) && !_.isNil(input.itemId)) {
    // Labels of an item
    filter['itemIds'] = input.itemId;
  }

  return filter;
}

function getItemAggregationPipeline(input: ItemsInput, getCount = false) {
  const matchQuery = {};
  const groupQuery = { _id: 0, itemIds: { $push: '$itemIds' } };
  const reduceOperator = {};
  let initialValue;

  if (!_.isNil(input) && !_.isNil(input.labelIds)) {
    // Items matching all labelIds
    const standardizedLabelIds = _.map(input.labelIds, standardizeLabel);
    matchQuery['id'] = { $in: standardizedLabelIds };
    matchQuery['pending'] = { $exists: false };
    groupQuery['initialSet'] = { $first: '$itemIds' };
    initialValue = '$initialSet';
    reduceOperator['$setIntersection'] = ['$$value', '$$this'];
  } else {
    // No label filter
    initialValue = [];
    reduceOperator['$setUnion'] = ['$$value', '$$this'];
  }

  const pipeline: any = [
    { $match: matchQuery },
    {
      $group: groupQuery
    },
    {
      $project: {
        itemIds: {
          $reduce: {
            input: '$itemIds',
            initialValue: initialValue,
            in: reduceOperator
          }
        }
      }
    }
  ];

  if (getCount) {
    pipeline.push({ $project: { count: { $size: '$itemIds' } } });
  }

  return pipeline;
}

function resolvers(db: ClicheDb, _config: LabelConfig): IResolvers {
  const labels: Collection<LabelDoc> = db.collection('labels');

  return {
    Query: {
      label: async (_root, { id }) =>
        await labels.findOne({ id: standardizeLabel(id) }),

      items: async (_root, { input }: { input: ItemsInput }) => {
        const res = await labels
          .aggregate(getItemAggregationPipeline(input))
          .toArray();

        return res[0] ? res[0].itemIds : [];
      },

      itemCount: async (_root, { input }: { input: ItemsInput }) => {
        const res = await labels
          .aggregate(getItemAggregationPipeline(input, true))
          .next();

        return res ? res['count'] : 0;
      },

      labels: async (_root, { input }: { input: LabelsInput }) => {
        return await labels.find(getLabelFilter(input));
      },

      labelCount: (_root, { input }: { input: LabelsInput }) => {
        return labels.countDocuments(getLabelFilter(input));
      }
    },

    Label: {
      id: (label: LabelDoc) => label.id,
      itemIds: (label: LabelDoc) => label.itemIds
    },

    Mutation: {
      addLabelsToItem: async (
        _root, { input }: { input: AddLabelsToItemInput },
        context: Context) => {
        const labelIds = _.map(input.labelIds, standardizeLabel);
        const updateOp = { $push: { itemIds: input.itemId } };
        const errors = await Promise.all(_.map(labelIds, async (id) => {
          try {
            // cannot use updateMany because we need to upsert labels
            await labels.updateOne(context, { id }, updateOp, { upsert: true });

            return undefined;
          } catch (err) {
            console.error(err);

            return err;
          }
        }));
        if (errors.filter((err) => !!err).length === 0) {
          return true;
        }
        const errMsg = _.reduce(errors, (prev, curr, index) => {
          if (!curr) {
            return prev;
          }
          const delimiter = index ? ', ' : '';

          return `${prev}${delimiter}${labelIds[index]}`;
        }, 'Could not add the following labels to the item: ');
        throw new Error(errMsg);
      },

      createLabel: async (_root, { id }, context: Context) => {
        const labelId = id ? standardizeLabel(id) : uuid();
        const newLabel: LabelDoc = { id: labelId };

        return await labels.insertOne(context, newLabel);
      }
    }
  };
}

const labelCliche: ClicheServer = new ClicheServerBuilder('label')
  .initDb(async (db: ClicheDb, config: LabelConfig): Promise<any> => {
    const labels: Collection<LabelDoc> = db.collection('labels');
    await labels.createIndex({ id: 1 }, { unique: true, sparse: true });
    await labels.createIndex({ id: 1, itemIds: 1 }, { unique: true });
    if (!_.isEmpty(config.initialLabelIds)) {
      return labels.insertMany(EMPTY_CONTEXT,
        _.map(config.initialLabelIds, (id) => ({ id: id })));
    }

    return Promise.resolve();
  })
  .actionRequestTable(actionRequestTable)
  .resolvers(resolvers)
  .build();

labelCliche.start();
```

Figure 24: The server-side code that interfaces with the database for the Label cliche.

A.2. show-labels Cliché Action

```html
<ul *ngIf="_labels && _labels.length > 0" class="list-group">
  <li *ngFor="let label of _labels" class="list-group-item">
    <dv-include
      [action]="showLabel"
      default-showLabel="{ tag: label-show-label }"
      [inputs]="{label: label}"
      [parent]="showLabels">
    </dv-include>
  </li>
</ul>

<div *ngIf="!_labels || _labels.length === 0">
  <span>{{noLabelsToShowText}}</span>
</div>
```

Figure 25: The HTML code for the show-labels cliché action. Users can replace the show-label action with another cliché or app action.

```typescript
import {
  AfterViewInit, Component, ElementRef, EventEmitter, Inject, Input, OnChanges,
  OnInit, Output, Type
} from '@angular/core';
import {
  Action, GatewayService, GatewayServiceFactory, OnEval, RunService
} from '@deja-vu/core';
import * as _ from 'lodash';

import { ShowLabelComponent } from '../show-label/show-label.component';

import { API_PATH } from '../label.config';
import { Label } from '../shared/label.model';

interface LabelsRes {
  data: { labels: Label[] };
  errors: { message: string }[];
}

@Component({
  selector: 'label-show-labels',
  templateUrl: './show-labels.component.html',
  styleUrls: ['./show-labels.component.css']
})
export class ShowLabelsComponent implements AfterViewInit, OnEval, OnInit,
  OnChanges {
  // Fetch rules
  @Input() itemId: string | undefined;

  // Presentation inputs
  @Input() noLabelsToShowText = 'No labels to show';

  @Input() showLabel: Action = {
    type: <Type<Component>>ShowLabelComponent
  };

  @Output() labels = new EventEmitter<Label[]>();

  _labels: Label[] = [];

  showLabels;
  private gs: GatewayService;

  constructor(
    private elem: ElementRef, private gsf: GatewayServiceFactory,
    private rs: RunService, @Inject(API_PATH) private apiPath) {
    this.showLabels = this;
  }

  ngOnInit() {
    this.gs = this.gsf.for(this. elem);
    this.rs.register(this.elem, this);
  }

  ngAfterViewInit() {
    this.load();
  }

  ngOnChanges() {
    this.load();
  }

  load() {
    if (this.canEval()) {
      this.rs.eval(this.elem);
    }
  }

  async dvOnEval(): Promise<void> {
    if (this.canEval()) {
      this.gs.get<LabelsRes>(this.apiPath, {
        params: {
          inputs: JSON.stringify({
            input: {
              itemId: this.itemId
            }
          }),
          extraInfo: { returnFields: 'id' }
        }
      })
        .subscribe((res) => {
          this._labels = res.data.labels;
          this.labels.emit(this._labels);
        });
    }
  }

  private canEval(): boolean {
    return !!(this.gs);
  }
}
```

Figure 26: The TypeScript file for the show-labels cliché action. Users can declare inputs and outputs to the cliché action in addition to those for the requests to the server.