Designing Cliché Authorship in the Déjà Vu Web Development Platform

by

Ma. Czarina Angela Lao

S.B., Massachusetts Institute of Technology (2018)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment 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 N. Jackson
Professor
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Designing Cliché Authorship in the Déjà Vu Web Development Platform

by

Ma. Czarina Angela Lao

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2019, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Applications, though they differ vastly in purpose, tend to have many fundamental concepts in common, such as the concepts of authentication, ratings, and events. These recurring concepts are what we call clichés. Déjà Vu (DV) is a new platform that allows developers to build web applications without repeating cliché code, by assembling them using clichés from a library. While previous work on DV has focused on developing the cliché library and how clichés would be assembled to create applications, there has not been much work aimed at how the clichés themselves are written. Since clichés form the fundamental idea behind DV, it is vital that clichés can easily be created. This thesis explores and designs the underlying abstractions in DV to make the process of authoring clichés simple and straightforward. We successfully achieve this by constructing a DV command-line interface for scaffolding clichés; a `cliche-server` module which includes a database layer that hides the notion of transactions; and a subscription system which enables reactive clichés. These tools and libraries significantly reduce the code cliché authors need to write, as well as lessen cliché code complexity through the abstractions provided.

Thesis Supervisor: Daniel N. Jackson
Title: Professor

Acknowledgments

Thank you to Daniel Jackson, whose guidance and valuable insights helped shape this work.

Thank you to Santiago Perez De Rosso for working closely with me on this thesis, and for the lengthy discussions on various design and implementation ideas.

Déjà Vu is a product of the work of a number of people–my sincerest thanks goes to every single one of them, especially Santiago, Maryam Archie, and Barry McNamara. It has been a wonderful couple of years working on this exciting new platform with you.

To Maryam, many thanks for being the first user of the DV CLI. Your enthusiasm is unparalleled.

Thank you to my friends, 6.031 and the 6.031 staff, and all my previous mentors.

Lastly, my utmost gratitude goes to my family, for their constant love and support that knows no distance. Thank you for the extra sets of eyes on my drafts, and more importantly, for inspiring me in more ways than I can count. This thesis would not be the same without you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Websites have rapidly evolved from simply delivering static content to being dynamic applications for social media, online shopping, and team organization, to name a few. To support the variety of functionalities needed, code and software design have become more complex. Notice that though the aforementioned web application types serve different overall purposes, they share similar concepts. For example, a social media site can be broken down into the concepts of posts, friends, comments, upvotes, and profiles. Comments and profiles are similarly applicable to online shopping sites and forums. Currently, web developers need to rewrite programming logic for the same concepts across multiple web applications, while merely needing slight modifications for their specific use case. This needlessly consumes developer time and adds an additional source of complexity and bugs. To make the situation worse, common implementation mistakes that have already been solved before, both in terms of correctness and usability, continue to recur.

Déjà Vu (DV) is a web development platform that aims to address the aforementioned issues on design, correctness, and usability by encapsulating concepts into self-contained, reusable modules. DV draws on Daniel Jackson's notion of conceptual software design; that is, software design in terms of *concepts*, which he defines as building blocks of software systems [2]. He also asserts that bugs in software are often

due to underlying flaws in conceptual design [3]. By providing a set of well-designed, prepackaged concepts, DV enables developers to create web applications that are safe from bugs and usability issues due to errors in the design or the implementation of concepts.

In DV, concepts are represented by modules called *clichés*. Each cliché contains the full-stack implementation of the concept it embodies, from the frontend interface shown to users, to the backend data storage. The clichés in DV's catalog can then be assembled through the DV language to provide end-to-end functionality for web applications.

1.2 Shaping the Building Blocks

While DV as it currently stands already provides a robust collection of clichés, it is still far from being able to capture all the concepts that web applications may contain. Concepts also evolve and new ones are invented continually [3]. As such, DV will rely on a community of experts to enrich its library with more clichés. Since clichés form the backbone of DV-made web applications, it is vital that the clichés themselves are well-crafted modules. This thesis focuses on designing how to write clichés in a simple and straightforward manner to make it easy for developers to contribute to the cliché catalog. In line with the goals of DV as a whole, the effort and complexity involved in writing a cliché should ideally be on par with building similar functionalities in a regular web application. The work in this thesis provides an easy way for experts to author clichés through the construction of a set of libraries and tools tailored to aid cliché development.

1.3 Contributions

This thesis makes the following contributions:

- The Déjà Vu Command-Line Interface or DV CLI, which allows users to automatically scaffold clichés and add components to them;

- The `cliche-server` module, a module that provides the base and the API for every cliché's server;
- A database layer that encapsulates and handles transactions across multiple MongoDB database connections, necessary for atomic cliché operations; and
- A subscription system for DV that enables cliché authors to write reactive user-facing components, which can automatically update based on any changes to persisted data.

Chapter 2 provides an overview of Déjà Vu, introducing the various modules and considerations involved when authoring clichés for DV.

Chapter 3 presents the main work of this thesis, which are the DV CLI, the `cliche-server` module, including a database layer that abstracts away transactional logic, and the DV subscription system that enables reactive cliché components.

Chapter 4 describes the process we undertook to create a new cliché from scratch using the newly-built tools and libraries. We then evaluate the cliché authorship process based on this experience and prior ones.

Finally, Chapter 5 offers suggestions for future work and a summary of the work done.

Chapter 2

Déjà Vu Overview

This chapter provides an overview of Déjà Vu and the relevant ideas needed to understand cliché authorship in Déjà Vu. These include DV terminology, DV's structure, its components and modules, including the clichés themselves, and how they interact with one another.

2.1 Cliché Catalog

The main component of DV is its catalog, which is the library of concepts or clichés that it offers. Appendix A gives the complete list of clichés currently in the catalog along with their respective descriptions.

2.1.1 Cliché Examples

We describe a few select clichés that will be used to illustrate other ideas and terms in the succeeding sections. The clichés in the catalog can be categorized into three classes. The first involves security and includes the *Authentication*, the *Authorization*, and the *Passkey* clichés. These clichés internally do not function differently from other clichés, but are special in terms of the purpose that they serve. In addition to the concepts of authentication and authorization that they represent, they help enforce security in DV (see Section 2.3.2). The second class is where most clichés are classified

as they each represent a single concept, such as *Scoring* and *Task*, with predetermined data models. Finally, the last class, currently only satisfied by the *Property* cliché, is different from other clichés in the sense that the objects that it stores could have arbitrary schemas. The schema used by the *Property* cliché is specified by the DV web app developer through a configuration file. The succeeding paragraphs talk about the *Authentication*, *Scoring*, and *Property* clichés to explain each cliché class by example.

The *Authentication* cliché's purpose is to "verify the user's identity with a username and a password" and as such, it stores `User` objects in its backend. Each `User` has three fields–`id`, `username`, and `password`. The *Authentication* cliché utilizes those fields to perform authentication-related actions such as generating tokens out of `User` `id`s. Its actions, which will be discussed in the next section, can be used in conjunction with other actions to ensure that the user who triggered those actions is authenticated.

The *Scoring* cliché, as the name suggests, is used to keep track of scores. To do so, it contains `Score` objects that have the fields `id`, `value`, `targetId`, and `sourceId`. Notice that the source and the target of the score are stored in the form of IDs. The use of IDs allows cliché data entities to easily reference one another. For example, a `Score` object's `targetId` could be equivalent to a `User` object's `id`, indicating that the score is for the user being referenced. Similarly, the `sourceId` could also be equal to another *User*'s `id`, demonstrating the idea of one user giving a score to another.

As mentioned previously, the *Property* cliché is distinct from other clichés because it has a dynamic schema specified through a configuration file. The *Property* cliché exists for properties of objects that do not have a special function in the app, unlike passwords that need to be hashed in authentication, or score values that could be aggregated and ranked by target. Only the four basic operations, create, read, update, and delete (CRUD), are available and are needed for object properties under the *Property* cliché.

2.2 Cliché Actions

Each cliché also has a set of *actions* associated with it. Each action embodies the full-stack implementation of an operation related to that cliché and its data entities. For example, in order to perform the CRUD operations on the `User` object of the *Authentication* cliché, there are `register-user`, `show-user`, `update-user`, and `delete-user` actions.

Each action has HTML code, client-side JavaScript that calls the corresponding cliché's backend, and optional CSS code for styling.

An action's JavaScript code is responsible for making the appropriate request(s) in order to either fetch and display data, or mutate server-side data. We say that a cliché action has *evaluated* when it makes the request to read cliché data, and it has *executed* when it mutates or produces a side-effect on stored cliché data. This distinction ensures that the actions are RESTful [1], similar to how web frameworks differentiate between `GET` and `POST` HTTP requests. A cliché action typically evaluates when the action first loads, while it usually executes due to some end-user action, such as a button press on a Submit or a Login button, for example. It could also be executed indirectly through other cliché actions, which will be explained in the next section on cliché composition.

2.3 Cliché Action Composition

To make it easier to explain cliché action composition, we will use Slacker News (SN)[1], a web forum where users can register, log in, submit posts, upvote posts, and reply to posts. In terms of data entities, SN would need a `Post` object that has a `title`, a `url`, an `author`, and a `score`. It also needs `User`s where each one has a `username` and a `password`. Note that the author of a post is also a user in this web app, and as such, this is a situation in which clichés need to work hand-in-hand. In order to represent the SN data entities using DV, one would use the *Authentication* cliché for users; the *Property* cliché to store the `title`, `url`, and `author` fields of posts, where the `author`

[1]Slacker News is a simplified clone of Hacker News (https://news.ycombinator.com/)

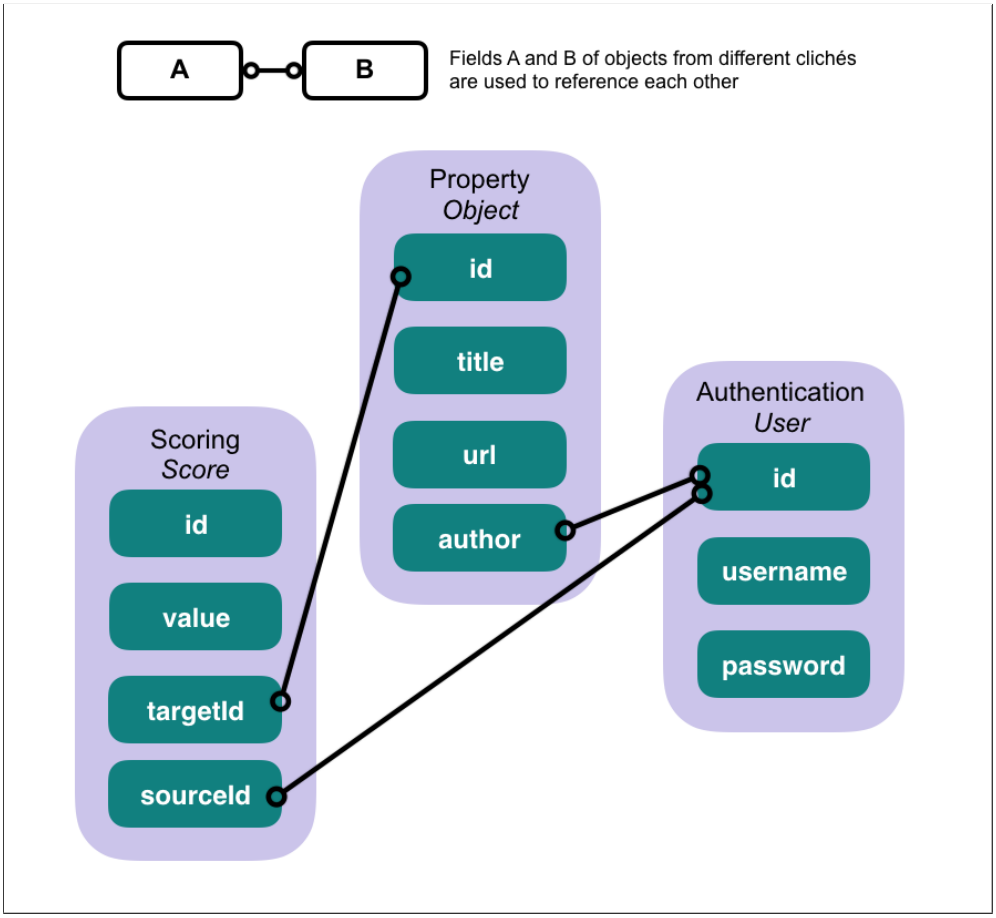


Figure 2-1: SN's data entities and their relationships

field is the `id` of the corresponding `User` in *Authentication*; and the `Scoring` cliché to keep track of each post's score. Each `Score` in *Scoring* would have a `targetId` that points to an `id` in *Property*, a `sourceId` to an `id` in *Authentication*, and a value of 1 to represent a single upvote. Each post would also need a starting score of 0, so a `Score` object with a value of 0 would also be created when a post is made. Figure 2-1 illustrates SN's data entities and the relationships described.

Cliché actions are the components that web developers use to assemble each page of web applications such as SN. For instance, the login/signup page of SN contains the `sign-in` and `register-user` actions of the *Authentication* cliché as shown in Figure 2-2. In this case, there is no special relation or coordination that happens between the two cliché actions–they are simply displayed next to each other.

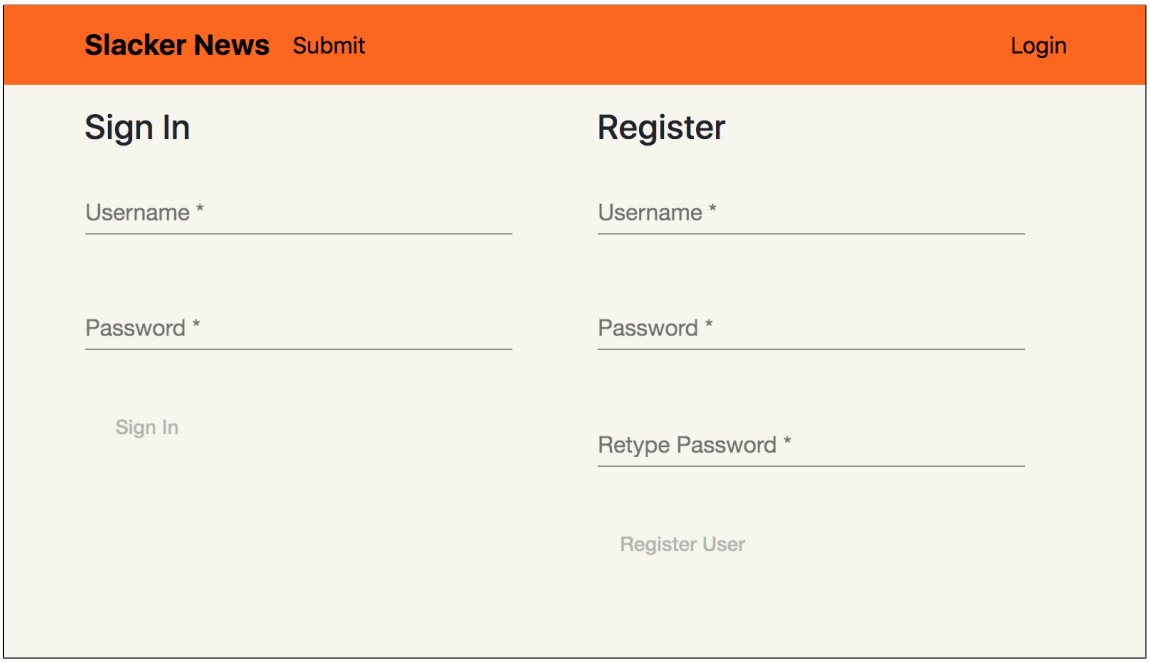


Figure 2-2: The login page of Slacker News, made using Déjà Vu

```
<dv.tx>
  <dv.gen-id />
  <authentication.authenticate id=sn.navbar.loggedInUser.id hidden=true />
  <property.create-object
    id=dv.gen-id.id
    buttonLabel="Submit"
    initialValue={ author: sn.navbar.loggedInUser.username }
    newObjectSavedText="Post submitted"
    showExclude=['author'] />

  <scoring.create-score targetId=dv.gen-id.id value=0 hidden=true />
  ...
</dv.tx>
```

Figure 2-3: Excerpt of the `submit-post` action transaction

2.3.1 Action Inputs and Outputs

Each action in DV can have input and output values, and an output of an action can be used as an input to any number of actions. For example, when we create a post in SN, we can generate a random ID using the built-in helper DV action `gen-id` (Figure 2-3, line 2). The generated ID is provided as an output of the action which we use an input to both the `create-object` action of *Property* and the `create-score` action of *Scoring*. Inputs of actions can be set as HTML attributes of the action, and outputs can be accessed through properties of the action. Lines 5 and 11 of Figure 2-3 show the `id` input of `create-object` and the `targetId` input of `create-score` both being set to `id` output of `gen-id` through `dv.gen-id.id`.

2.3.2 Atomic Operations

Cliché actions can also be composed on a web page such that they evaluate or execute in a single transaction. When one of the actions evaluates or executes, the others evaluate or execute too; and when one action fails, the entire operation fails. When a set of actions either all occur or all fail, the operation is said to be *atomic*.

In order to specify that cliché actions should form an atomic operation, web developers should surround the actions with `<dv.tx>` tags, similar to the way HTML tags are used, as the `submit-post` excerpt of SN in Figure 2-3 demonstrates. Atomic operations are necessary in Déjà Vu so that users do not observe data in an inconsistent state. Since DV may use multiple clichés to represent a single entity in an application, the execution of a set of cliché actions can be conceptually equivalent to a single operation on an entity in the app. For instance, in SN, when a user submits a post, the creation of all the objects related to it must be atomic. Recall that a SN post is represented by an object in each of the *Property* and the *Scoring* clichés. In the code excerpt in Figure 2-3, the actions included are `gen-id` to generate an ID for the new post, `authenticate` from *Authentication*, `create-object` from *Property*, and `create-score` from *Scoring*. Once the Submit button on `create-object` is pressed, all the actions inside the `<dv.tx>` tags are executed. All these executions

would then be run in a single transaction using a two-phase commit (see Section 2.4). If the creation of the `Score` object in the *Scoring* cliché fails, then the corresponding object in the *Property* cliché will not be created either. SN will then report that the post submission failed. We do not want only the object in *Property* to be created, for example, because then, SN would have posts that do not have a score.

Security through Atomicity

DV also makes use of atomic operations for security. Using the same example shown in Figure 2-3, to ensure that a user submitting a post is authenticated, the web app developer should include the `authenticate` action from the *Authentication* cliché in the same transaction that creates the relevant objects in *Scoring* and in *Property*. This way, if the user were not genuinely authenticated, the `authenticate` action would fail, causing the entire operation to fail.

2.4 System Architecture

Internally, the DV backend modules communicate using the two-phase commit (2PC) protocol to support transactions among multiple MongoDB database connections. As demonstrated by Figure 2-4, requests originate from each individual cliché action. Requests get processed, batched by transaction, and sent by DV's client-side runtime system to the server-side gateway. For each transaction, DV's gateway module initiates 2PC with the servers of the relevant clichés. In addition to the original request information, the gateway includes the 2PC phase being executed in its message to each cliché server. Each cliché server should thus have programming logic to handle each phase of the protocol accordingly.

2.4.1 Two-Phase Commit Protocol

The two-phase commit protocol is a standard coordination scheme that enables operations on multiple databases to be performed in a transaction. In the protocol, there is a coordinator that sends and receives messages to and from the participating

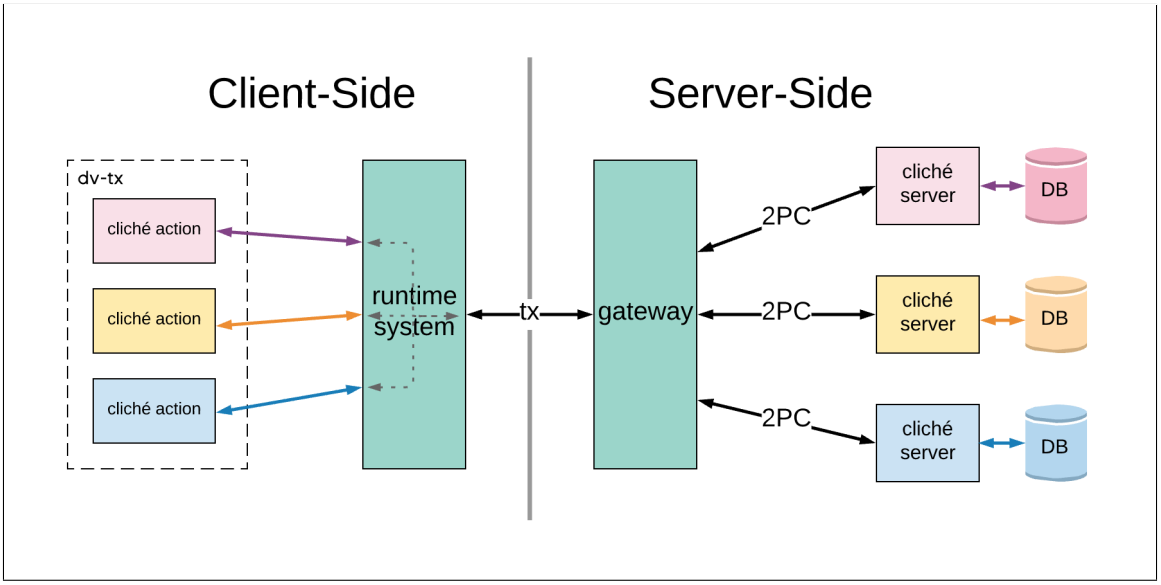


Figure 2-4: System architecture of Déjà Vu

databases. The first phase of 2PC is the voting phase. In this phase, the coordinator sends a `vote` message to all participants about a request, then waits for a "yes" or a "no" vote from each one. Each participant verifies whether it can successfully execute the request or not, and replies with the appropriate "yes"/"no" message. The coordinator transitions to the completion phase once it can decide whether the transaction can be committed or not. If it receives a "yes" from every participant, the transaction can be completed and the coordinator sends a `commit` message to the participants. The coordinator can also enter the completion phase once it receives the first "no" response. In this case, the coordinator sends an `abort` message to each participant. The coordinator could also send an `abort` message if it times out waiting for replies from the participants.

In DV, the gateway module acts as the 2PC coordinator and the servers of the clichés involved in the transaction are the participants. Cliché servers decide whether to vote "yes" or "no" based on the validity of the request and their ability to successfully execute the request. The reasons they could vote "no" include:

- **Client Error.** The request sent by the client is invalid or cannot be performed successfully. These are standard errors that happen in a normal web application

depending on the end-user's inputs. Some possible causes of this kind of error are:

  - Executing the request could violate database constraints, such as a constraint on the uniqueness of IDs. For instance, when users register for an account in a website, they could try to use a username that is already taken.
  - The object involved in the request does not exist. Requests in DV typically define the object on which to perform an operation through an ID. If there is no object in the database that has that ID, there is no way to successfully update or delete that non-existent object.

  This type of error is equivalent to the `4xx` error status code in HTTP, where the client should not retry the request because it will likely fail again, unless for example, the object that caused the error is deleted.

- **Server Error.** The gateway can process multiple transactions simultaneously. As such, when the cliché server handles a new request on an object, there can still be pending changes on the object due to another concurrent request. There cannot be multiple pending operations on an object because the execution of one could invalidate the other. Similar to the `5xx` status code in HTTP, since the error is not due to the client and is likely a temporary inability of the server to execute the request, clients should attempt to send the request again.

2.4.2 Cliché Servers

Since clichés must be self-contained, every cliché needs its own backend, the cliché server, which contains the transaction-handling code and the separate connections into the MongoDB database. The server needs to handle each type of message sent by the gateway (`vote`, `commit`, and `abort`), and respond to the gateway as required by the 2PC protocol. Other database-related responsibilities include defining the schema of the objects to be stored in the database, the queries and the mutations

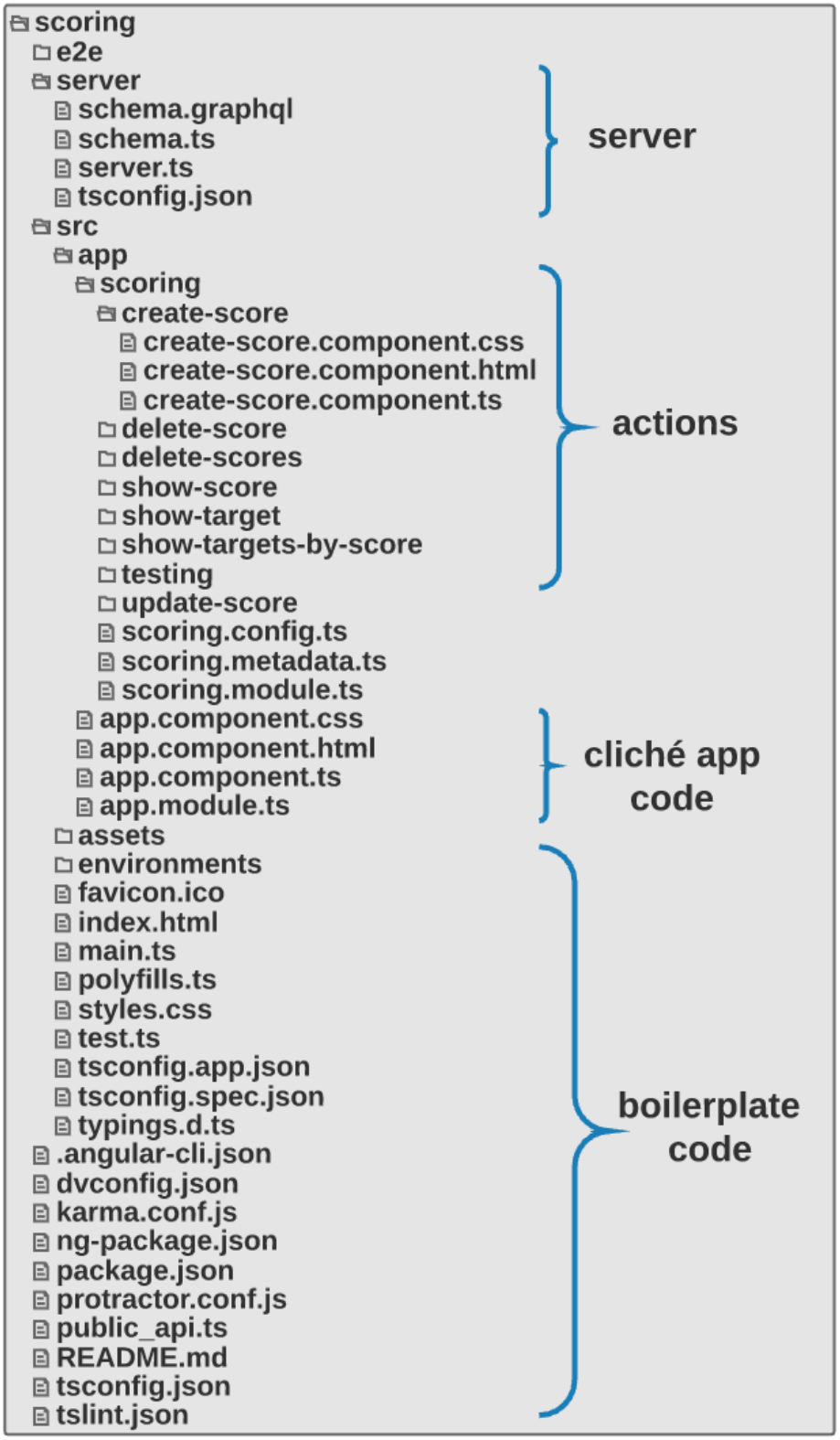


Figure 2-5: Folder structure of a cliché

that clients of the database can use, and which query and mutation requests are for which cliché action(s).

The cliché server is also responsible for exposing the HTTP endpoints of the cliché so that the gateway can communicate with it. The gateway provides the 2PC message type, the name of the cliché action from which the request originated, and some request metadata. The server determines and executes the appropriate database operations based on the provided request information. The gateway is configured so that it knows which endpoints to use for which clichés.

2.5 Cliché Package Structure

To summarize the cliché components discussed in this chapter, we show the package structure of a cliché in Figure 2-5. Cliché developers would need to come up with such a package in order to author a cliché. In the figure, the `server/` folder consists of files that declare the schema both in terms of GraphQL and TypeScript, and the `server.ts` file which defines the corresponding database operations for every cliché action. The `src/` folder contains the client-side code for each cliché action. Instead of having to use the cliché actions in a web application, cliché authors are able to preview and test actions through the cliché app. The cliché app is part of the cliché package and is defined by the `app.component` files. The rest of the files are various configuration files, test files, and boilerplate code.

Chapter 3

Cliché Authorship Tools

In this chapter, we present the tools and libraries built to simplify the process of authoring clichés. We start with the DV command-line interface for code scaffolding, then proceed to describe the `cliche-server` module, the database layer, and DV's subscription system.

3.1 Cliché Command-Line Interface

Since each Déjà Vu cliché is an Angular[1] project, it requires all the necessary scaffolding an Angular project needs as well. Angular provides a command-line interface (CLI)[2] to scaffold an Angular project and to add more components into one. For example, with the `ng new` command, the Angular CLI can automatically generate the folder structure and files shown in Figure 3-1. In addition to the Angular CLI, other popular web development frameworks such as React[3] and Vue.js[4] also offer the same kind of CLI tooling. Online comparisons of web frameworks have tooling as a criterion and emphasize that developers give importance to being able to automate tasks that could be categorized as "grunt work", such as initializing projects and scaffolding new components [11]. DV clichés have a significant amount of boilerplate code that could

[1]Angular web development framework: `https://angular.io/`
[2]Angular CLI: `https://angular.io/cli/`
[3]React: `https://reactjs.org/`
[4]Vue.js: `https://vuejs.org/`

Figure 3-1: Folder structure of a new Angular project

automatically be generated as well. As such, it is only practical that DV utilizes the Angular CLI to allow cliché authors to also easily begin with the necessary files and folders.

Instead of simply building a thin layer on top of the Angular CLI, we've built the DV CLI using Angular's Schematics [5], a template-based code generator on which the Angular CLI itself relies. Schematics allows the DV CLI to generate an Angular-structured project customized for Déjà Vu. Figure 3-2 shows excerpts from the `package.json` file that the Angular CLI would normally generate versus the file that the DV CLI creates.

[5] Angular's Schematics: `https://angular.io/guide/schematics`

```
{
  "name": "my-project",
  "version": "0.0.0",
  "license": "MIT",
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build --prod",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e"
  },
  "private": true,
  "dependencies": {
    ...
  },
  "devDependencies": {
    "@angular/cli": "~1.7.4",
    "@angular/compiler-cli": "^5.2.0",
    ...
  }
}
```

(a) Standard Angular `package.json` file

```
{
  "name": "@deja-vu/comment",
  "version": "0.0.1",
  "scripts": {
    "start": "dv serve",
    "build": "npm run
      dv-build-comment",
    "package": "dv package",
    ...
    "ng": "ng",
    "dv": "dv"
  },
  ...
  "devDependencies": {
    "@angular/animations": "^5.2.0",
    "@angular/cdk": "^5.2.1",
    "@angular/cli": "^1.7.3",
    ...
    "@deja-vu/core": "0.0.1",
    "@deja-vu/gateway": "0.0.1",
  }
}
```

(b) DV cliché `package.json` file

Figure 3-2: Comparing excerpts of files made by the Angular CLI and the DV CLI

3.1.1 Commands

The two main commands that the DV CLI provides are

```
dv new cliche <name>
```

to scaffold a new cliché with the given name, and

```
dv new action <type> <entityName> [actionName]
```

which creates a new cliché action based on the provided type, entity name, and action name. The argument `type` should be one of `create`, `show`, `update`, `read`, or `blank`, and `[actionName]` is an optional parameter. When `actionName` is not given, by default the action name is `<type>-<entityName>`, taken from the other two required parameters. The complete list of DV CLI commands are in Table 3.1.

Command	Description
`dv get <key>`	Get a value from the configuration
`dv new <type>`	Create a new cliché or action
`dv new cliche <name>`	Create a new cliché
`dv new action <type> <entityName> [actionName]`	Create a new action with the given type, entity name, and an optional action name
`dv package`	Package a cliché
`dv completion`	Generate the bash completion script
`dv serve`	Start the server for the cliché app or DV app

Table 3.1: DV CLI Commands

The `dv new cliche foo` command creates a new cliché `foo` with a single data entity `foo`; the HTML, CSS, and TypeScript files for four starting actions, `create-foo`, `show-foo`, `update-foo`, and `delete-foo` to represent the CRUD operations on `foo`; the server-side code to handle the database queries and mutations that relate to the provided cliché actions; the `app.component.html` file to preview and test cliché actions (Figure 3-3) with the starting actions included in the file; and the DV-customized Angular and TypeScript configuration files and boilerplate code. The `dv new action` command creates the HTML, CSS, and TypeScript files for the new action and adds the action to the `app.component.html` file.

3.1.2 Code Standardization and Example Code

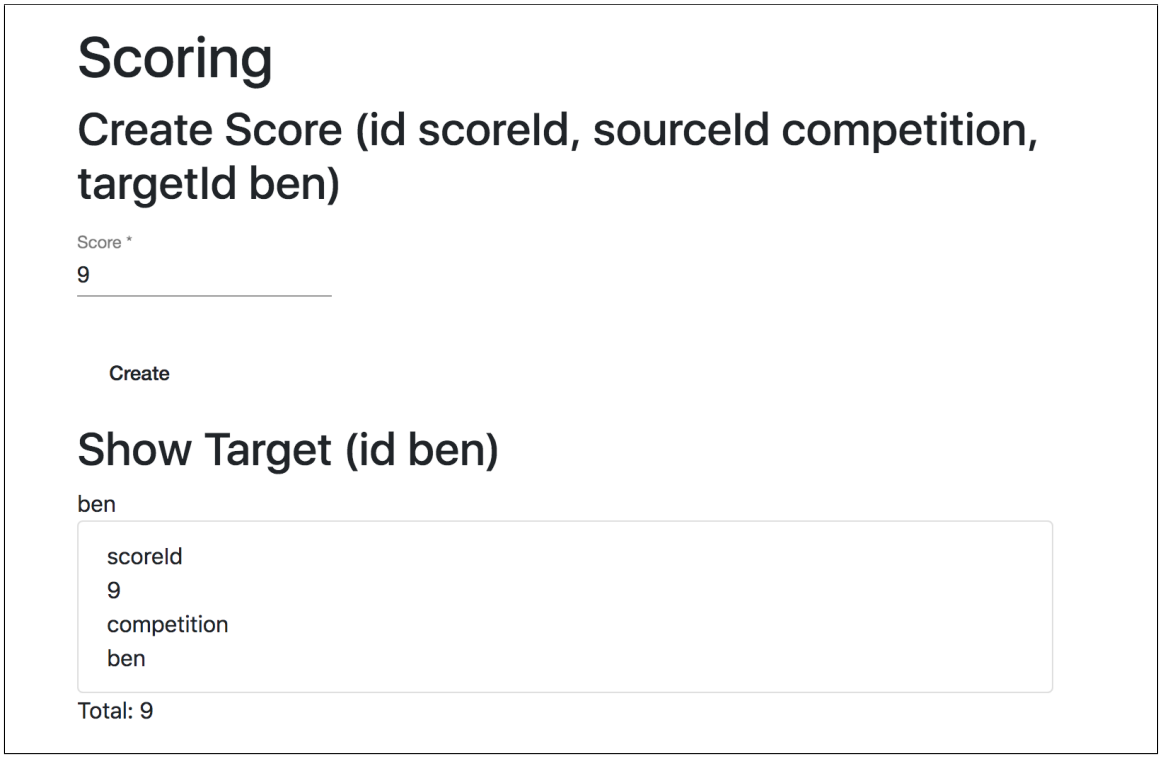
There are a number of other advantages and opportunities that the code scaffolding of the DV CLI brings. First, this is an opportunity to provide sample or starter code. It has been shown that the availability of extensive code samples drives developer adoption of new programming languages [5], so we analyzed the 18 existing clichés in the catalog to determine common use patterns that cliché authors would need. While the existing clichés themselves are a good set of code samples, we went a step further by including the code samples that would most likely be needed in the starting code of a cliché. Providing example starter code also gives the benefit of automatically

```
<div class="container">
  <h1>Scoring</h1>

  <h2>Create Score (id scoreId, sourceId competition, targetId ben)</h2>
  <scoring-create-score
    id="scoreId"
    sourceId="competition"
    targetId="ben">
  </scoring-create-score>

  <h2>Show Target (id ben)</h2>
  <scoring-show-target id="ben"></scoring-show-target>
</div>
```

(a) Excerpt of the *Scoring* cliché `app.component.html` file



(b) Developer preview of the *Scoring* cliché's actions

Figure 3-3: The HTML code and the resulting web page for the *Scoring* cliché app

setting the conventions and standards when writing cliché code. We would have to preserve the quality of the clichés in the catalog and it would help to make it easy for developers to do so. These are the findings that motivated what code the `dv new cliche` command generates:

- All the data entities in the clichés need CRUD operations or similar. As such, clichés start with the full-stack implementations of the actions `create-foo`, `show-foo`, `update-foo`, and `delete-foo`, where `foo` is the entity with which the cliché starts.
- 15 out of the 18 clichés have a single data entity, while the remaining three have two, so it makes sense to only provide one set of CRUD actions.
- 10 out of 18 of the clichés have a data entity which carries the same name as the cliché. That is, if the cliché is named `foo`, the entity it has is also called `foo`. The DV CLI thus uses the cliché name as the name of the entity for the starting actions it provides.

3.1.3 A Fully Functional Cliché

We've designed the DV CLI so that after running a command to scaffold a new cliché, the cliché can immediately be built and run. In this way, cliché authors have an immediate idea of how the they work and what they look like. For example, executing the command `dv new cliche comment` produces all the code needed for the *Comment* cliché with actions `create-comment`, `show-comment`, `update-comment`, and `delete-comment`. Once the code for the fully-functioning cliché is generated, the cliché author can run `dv serve` to build and run the cliché. Visiting the web page at `http://localhost:3000/` then yields the result shown in Figure 3-4.

3.2 Cliché Server Module and Database Layer

In order for clichés to truly be independent components from one another and encapsulate the relevant concept, each cliché has to have its own backend server and

Figure 3-4: Cliché action preview page of a newly-generated cliché

database. While the servers have to be separate, we can still identify and encapsulate non-cliché-specific server code. We continue Déjà Vu's method of decomposition into modules [8] and have created the DV `cliche-server` module that cliché authors can use to implement a cliché's server. The `cliche-server` module abstracts away the basic API that cliché servers need: the endpoints with which the DV gateway communicates, and each cliché's database. This abstraction also makes general cliché server code easily modifiable. For example, if we needed to expose more endpoints for each cliché, it would be more effortless to solely change the `cliche-server` module, instead of all the clichés, which is precisely the goal of abstraction.

Since clichés do not share the same database, atomic operations among clichés are not as straightforward to apply as on a single database. Normally, a single database would natively support transactions, which gives atomicity of operations. In order to add support for transactions in Déjà Vu, we've previously implemented a two-phase commit (2PC) protocol between the DV gateway and the cliché servers as mentioned in Section 2.4. Cliché servers are thus required to be aware of the use of 2PC and

perform database operations as appropriate. Note that only database operations that perform mutations use the 2PC protocol since those are the only methods that could cause the data to get to an inconsistent state. As part of this thesis, we've created a database layer that hides from cliché authors the notion and the implementation of transactions, lessening the amount and the complexity of code they would need to write.

3.2.1 MongoDB

The main technologies we've chosen to use in the `cliche-server` module are Node.js[6] and MongoDB[7]. MongoDB, the database which the module uses, is a database that stores data as objects that can have arbitrary fields. The objects in MongoDB are called *documents* and documents can be grouped into *collections* in a MongoDB database. The MongoDB Node.js Driver API[8] allows users to perform CRUD operations on a single document or on multiple ones. For example, there is an `updateOne` and an `updateMany` method. In order to determine on which document(s) to execute an operation, the API's methods take in a `filter` parameter. The `filter` parameter is an object that specifies properties to match the document(s) on which to perform the desired operation. For instance, a user could make the method call

```
collection.updateOne({id: 'alice'}, updateOperation)
```

to apply `updateOperation` on the document with the `id` field that has a value `alice`.

3.2.2 Database Layer API

The database layer API is patterned after the MongoDB Node.js Driver API to still give developers the sense of using MongoDB operations, without the need to know or worry about the fact that a 2PC is being executed. The main difference between

[6]Node.js: `https://nodejs.org/`
[7]MongoDB: `https://www.mongodb.com/`
[8]MongoDB Node.js Driver: `https://mongodb.github.io/node-mongodb-native/`

the database layer's methods and the original methods is that the former requires a `Context` object parameter to be passed in, which includes the field `reqId` for the ID of the request and the field `type` for the request type. The request type is based on what the 2PC coordinator wants the cliché server to do. Its value could be `vote`, `commit`, `abort`, or `undefined`. The value is `undefined` if the request is not being executed with the 2PC protocol and as such, does not belong in a transaction. The `cliche-server` module API would provide the `Context` object, so cliché authors need only pass that same object through to the database layer method calls. For example, the MongoDB `updateOne` method call in the previous section would become

```
collection.updateOne(context, {id:  'alice'}, updateOperation)
```

instead. Appendix B shows the complete list of methods that the database layer exposes and their method signatures. The list does not cover all the methods that the MongoDB Node.js Driver has since we've only implemented the ones that are currently used in the clichés. All the basic CRUD operations are included, as well as methods for aggregation.

3.2.3 Implementation Requirements and Assumptions

In order for 2PC to work correctly, the database layer has to follow the requirements of the protocol. The first is that the database layer should respond with a "yes" to a `vote` message only when it is certain that the request can be applied successfully, i.e. the request is valid. Second, in between the voting and the completion phases of a request, no changes should happen that would change the previously established validity of the request. This is to ensure that if a `commit` message arrives, the operation would not fail. In the clichés Déjà Vu currently offers, we've identified that request validity can be determined based solely on the value of the document on which the request is being performed. This means that once a request that affects a certain set of MongoDB documents is deemed valid, only changes to that set of documents can invalidate the request. In our implementation, we assume this to be the case and

other request validity constraints are discussed in Section 3.2.10.

3.2.4 Locking Protocol

The use of locks is a common technique to create atomic operations. As a general approach, the database layer locks each MongoDB document to ensure that no changes happen to the document in between the voting and the completion phases of the 2PC. We use an additional boolean field on the MongoDB document called `_pending` to indicate whether the document has been locked or not. When the `_pending` field is set to `true`, it means that there are pending changes due to a request in progress. When it is `false` or does not exist as a field, no request possesses the document's lock.

We also need a way to indicate which request owns the lock, and what type of changes are pending on the document. The latter is essential for when the document is pending creation. This is so that the database layer can filter those documents from any query results it returns. We use another additional field to keep track of these details. The `_pendingDetails` field is an object field, which contains the subfields `reqId` and `type` based on a transaction's `Context`. The pending change could either be the creation of the document, an update to the document, or its deletion. If `_pending` is `false` or is unset, the `_pendingDetails` field should not exist as well.

We use the `_pending` field as a lock to make operations atomic. When an operation involves a change to a document, it must first obtain the lock to that document in order to proceed. Acquiring the lock is done by setting the field to `true`. MongoDB guarantees individual operations to be atomic for each document [6] so only one update will be successful when there are concurrent requests to acquire a document's lock through the `updateOne` command. Only the request that has successfully set a document's `_pending` field to `true` owns the lock to that document. MongoDB's `updateOne` method conveniently reports whether the `_pending` field was set to `true` due to the update or not. Once a request has the lock, it can apply as many updates as it wants on the document until it releases the lock. Requests that do not successfully obtain the lock to a document cannot make changes to the document and will receive

a `ConcurrentUpdateError`, which counts as a "no" vote. These unsuccessful requests can choose to retry as needed.

3.2.5 Voting Phase

In the voting phase of 2PC, we have to ensure that a requested change is possible. To do so, we need to check the validity of the request and acquire the lock of the document atomically. For the sake of simplicity, the succeeding discussions will mostly be in terms of the mutator methods that operate on a single document, `insertOne`, `updateOne`, and `deleteOne`. The procedures described below can easily be transformed for their counterparts that operate on multiple documents, `insertMany`, `updateMany`, and `deleteMany`, by replacing the use of method `foo` to `fooMany`. For example, in order to obtain a lock for some document(s) defined by the `filter` parameter, if we call

```
collection.updateOne(filter, { $set:  { _pending:  true } })
```

for a single document, then for multiple ones, it will be

```
collection.updateMany(filter, { $set:  { _pending:  true } }).
```

Note, however, that `updateMany` only guarantees the modification of each document that matches `filter` to be atomic, but the operation as a whole is not. We have decided to provide an atomicity guarantee on a per-document level and allow interleavings that could cause a document to no longer match `filter`, for example. Nevertheless, after the acquisition of the locks of multiple documents, assuming request validity is solely determined by each of those documents individually (Section 3.2.3), it can no longer be invalidated for that set of documents until the completion phase, as required by the 2PC protocol. Section 3.2.10 offers options that can make multi-document operations atomic. Any other situation wherein the procedure would be different for operations on multiple documents will be explicitly mentioned and the

difference explained.

insertOne

To insert a new document, we verify that it will satisfy uniqueness constraints imposed by the indices on the MongoDB collection of which the document is part[9]. Typically, the uniqueness constraint is on the document's ID, but other entities have uniqueness constraints on pairs of fields, such as `sourceId` and `targetId`, as in the *Scoring* cliché. The simplest way to do the check without possible interleavings is to try to insert the new document. If it is successful, then it satisfies the collection's uniqueness constraints and we can vote "yes". If not, then it doesn't and we vote "no" through a `DuplicateKeyError`. To acquire the document's lock at the same time as inserting it, we simply include the `{ _pending: true }` field in the document to be inserted. It should also include the `_pendingDetails` field when it performs the insert. Inserts should not set the `_pending` and `_pendingDetails` field separately to prevent interleaving operations that could potentially observe the document that is still pending creation. If a document only has the `_pending` field set, without the `_pendingDetails` field that indicates it is still pending creation, the database layer will fail to filter it out when providing query results. Database queries such as `find`, `findOne`, and `aggregate` filter out documents whose `_pending` field is true and `_pendingDetails.type` field is `create`.

updateOne and deleteOne

While we can insert a pending document and filter it out from query results in order to perform a validity check, we cannot apply a pending update to a document because the database should still be able to return the original one for queries that occur before the update is committed. It would be possible to have both the original version and the pending new version, but we found it unnecessarily more complicated to store pending versions of documents given the structure of existing clichés. In our implementation, we assume that uniqueness constraints only apply to immutable

[9]Section 3.2.10 covers other request validity constraints.

fields. As such, an update to a document will never violate these constraints. To update or to delete a document, we need only make sure the document exists. We can thus simply call an update operation on the document that sets its `_pending` field to `true`. MongoDB will then report whether the document was found or not, and whether any update was applied. We vote "yes" or "no" accordingly. If it was not found, the database layer throws a `NotFoundError`. If the document was found but the update was not applied, it means that the field was already `true`, so the method throws a `ConcurrentUpdateError`.

When trying to acquire the lock on multiple documents, we can decide when to throw a `ConcurrentUpdateError` by comparing the values of `matchedCount` and `modifiedCount`, which are both provided in the result object returned by MongoDB. If they are not equal, then the error should be thrown after releasing the locks on documents whose locks were acquired. Note that this procedure is simply a generalized version of the single document case. In the single document case, we throw a `ConcurrentUpdateError` when `modifiedCount` is 0, and since we've already verified that `filter` matches at least one document, we are sure that `matchedCount` > 0, which means this is the same as checking `matchedCount` $\neq$ `modifiedCount`.

Once the update or delete request has successfully obtained the lock, it should then perform another update to set the `_pendingDetails` field, which would include its request ID and type. We cannot perform this update at the same time as setting `_pending` to `true` because different requests would want to set the `_pendingDetails` field to distinct values which could overwrite each other when done concurrently. We want concurrent requests to perform exactly the same update, which is to only set `_pending` to `true`, so that MongoDB can properly report which request caused the update and which request(s) were redundant and thus unsuccessful in acquiring the lock. Figure 3-5 summarizes the procedure that happens in the voting phase of updates and deletes.

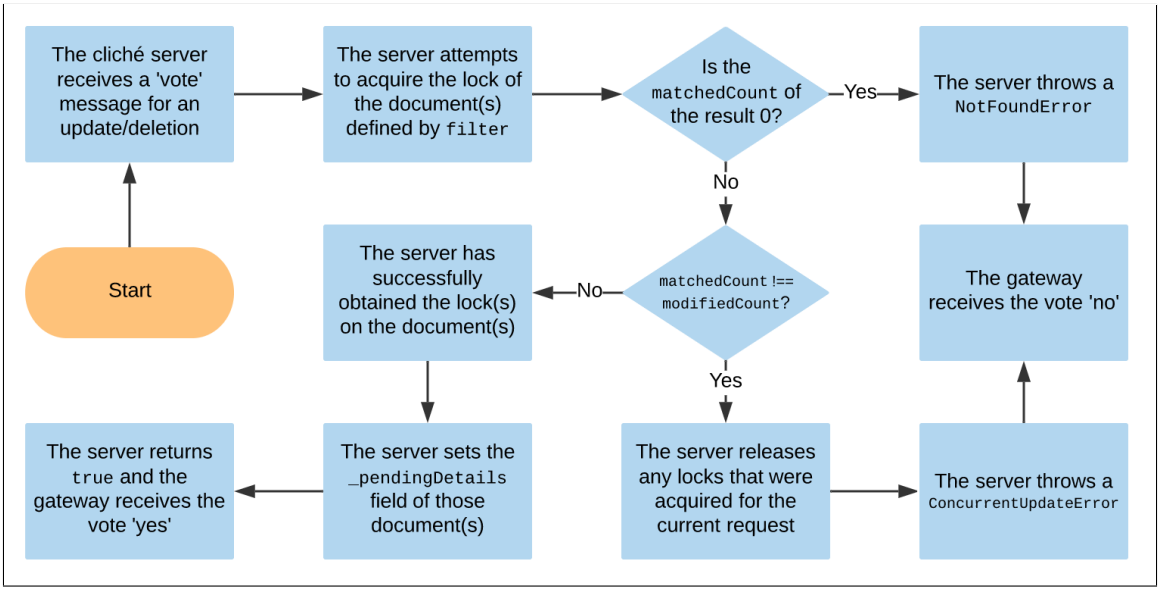


Figure 3-5: A flowchart of the events in the voting phase of an update or a delete operation

Upserts

MongoDB has standard methods for insert and update, but it allows users to insert new documents through the update operation as well. The `updateOne` method takes in an optional `updateOptions` object parameter that lets users specify the update option `upsert` as `true`. If the operation

```
collection.updateOne(filter, updateOperation, {upsert:  true})
```

does not find any documents that match the filter, MongoDB creates a new document based on the filter. The database layer update-handling scheme has to take this into account because as discussed under `insertOne`, we cannot allow interleaving operations between setting `_pending` to `true` and `_pendingDetails.type` to `create`. Fortunately, the database layer can take advantage of the special MongoDB `updateOperation` field `$setOnInsert`. If the update method results in an upsert, the value of the `$setOnInsert` field is set on the newly-created document. To acquire a document's lock for upserts, we specify

```
{ $setOnInsert:
  { _pendingDetails:  { type:  'create', reqId:  context.reqId } }}
```

in addition to

```
{ $set:  { _pending:  true }}
```

in order to handle upserted documents as if they were created using `insertOne`.

Transitioning to the Completion Phase

Once the cliché server has sent its "yes" or "no" vote to the gateway, it does not need to do anything until the next request from the gateway arrives. If the next message from the gateway is for the same transaction, the message type should now be either `commit` or `abort`. If the "yes" or "no" message to the gateway gets dropped, the gateway will time out and send an `abort` message to the cliché servers involved in the transaction. Other failure scenarios are currently not handled by DV yet.

3.2.6 Completion Phase: Commit

Responding to `commit` messages in the completion phase is straightforward. In this situation, we apply the pending changes on a document. For inserts, we simply remove the pending marker so that the document now appears in query results. We can also apply an update or a delete directly to the document. We are certain that the document has not changed since the lock was acquired in the vote phase because of the locking protocol described in Section 3.2.4. As such, the update or the delete should happen successfully.

3.2.7 Completion Phase: Abort

When we receive an `abort` message in the completion phase, we make sure any indication of pending changes are removed. For inserts, we simply delete the document and for updates and deletes, we release the lock by removing both the `_pending` and the `_pendingDetails` fields.

3.2.8 Non-Transactional Requests

Since requests that don't belong in a transaction could be sent concurrently with requests that are in a transaction, the former should respect the locking protocol and should not perform any updates on documents to which it does not own the lock. In other words, a non-transactional request should also be treated as transaction by itself. Otherwise, the former could apply updates to documents that have pending changes, breaking the concurrency design set by the locking discipline.

We follow the locking protocol by essentially executing the vote and the commit or the abort phases in succession. If the voting phase is successful, we proceed to the commit completion phase immediately after, and if not, we move to the abort completion phase. For new documents, those sequences of events are equivalent to simply inserting the new document into the database. For updates, we first obtain the lock to make sure we are allowed to perform the update, apply the update to the document if the lock was successfully acquired, then finally release the lock by unsetting the pending-related fields. If the lock was not successfully obtained, the update does nothing. We perform a similar procedure for deletes except that we would not need to remove the lock from the document if the operation is successful, as the document would no longer exist. This method is guaranteed to work alongside concurrent transactional requests because it follows exactly the same locking protocol.

3.2.9 Update or Deletion Preconditions

Sometimes, we may only want to perform an update to a document if a certain precondition is satisfied based on the document's current value. Without concerns about atomicity, what one can do is call `findOne` to obtain the document, check the precondition, then call `updateOne` if the precondition is satisfied. For example, in the *Transfer* cliché where money can be transferred from one account to another, we would first want to check whether an account has enough money to perform a transfer. Similarly, we may also want to perform an update based on the document's value. MongoDB's `updateOne` already natively supports certain preconditions and

updates that rely on the document's existing value through update operators such as `$max` (only updates a field if the specified value is greater than the existing field value) and `$inc` (increments the value of the field by the specified amount). However, MongoDB's native operators still do not satisfy all possible cases, such as what is necessary in the *Transfer* example.

To support the need for an atomic "check precondition then apply update" operation, we've created the `findOneAndUpdateWithFn` method. Since we've already introduced locks into the system, we can take advantage of them in order to make a sequence of method calls atomic. The `findOneAndUpdateWithFn` method is similar to `updateOne` except that it takes in an `updateFn` parameter instead of an `updateOperation`, and an additional optional `validationFn` parameter. `updateFn` is a function whose method signature is `T` $\rightarrow$ `UpdateOperation | undefined`, where `T` is the type of the document, the return type `UpdateOperation` is the specific object type for an update operation to be applied on a document, and returning `undefined` means that the document should be deleted. The `validationFn` parameter is the function that checks the precondition. It takes in a document of type `T` and does not return anything, but it should throw an exception if the document does not satisfy the precondition. This was done instead of having the function return `true` or `false` so that cliché authors can specify why the precondition was not met through the type of error that it throws.

The `findOneAndUpdateWithFn` method first attempts to acquire the lock of the document, and if it is not able to obtain the lock, it behaves similarly to `updateOne`. If it is successful, however, it calls `findOne` to get the document. It then calls `validationFn` on the fetched document to check the precondition, catching any error the validation function may throw. If it catches an error, it aborts the update by releasing the lock or deleting any upserts, and rethrows the error. If the precondition is satisfied, it votes "yes". If the request is not part of a transaction, or if the cliché server receives a `commit` message, the `updateFn` is called on the document to produce the `updateOperation` object to be applied (or `undefined` to signal a delete). Finally, `updateOne` is called with the produced `updateOperation` object and the lock

is released (or the document is deleted). Since the lock of the document was acquired as the first step, we are certain that there will be no interleaving operations on the document within the sequence of method calls.

3.2.10 Other Request Validity Constraints

The only constraints that can be enforced atomically for inserts are the uniqueness constraints set by the created indices. For updates and for deletes, we can check that the filter matches a document to update or delete. We can also use the `findOneAndUpdateWithFn` method described in the previous section.

Alternatively, clichés could handle transactions by themselves and not rely on the database layer. They could instead perform operations that involve multiple documents or multiple document collections in a transaction by using the method `ClicheDb.inTransaction()`. This method is the `cliche-server` module's thin wrapper around MongoDB's transaction mechanism within a single database connection. Note that this method can perform operations in a transaction only for a single phase of the 2PC protocol at a time. As such, it is still necessary to use the locking protocol in addition to the `ClicheDb.inTransaction()` method to atomically check the precondition and acquire the lock of a document. The use of MongoDB's transaction mechanism incurs a significant performance penalty [7] on top of the performance cost that 2PC brings, so it should be used only when needed.

3.2.11 Limitations

The 2PC protocol has inherent limitations. Chief of them is that the gateway, the coordinator of the protocol, could be a single point of failure in the system. If it fails, all the transactions in progress cannot continue until it is brought back up.

The assumptions made throughout this section also impose limitations on the possible operations the system can support. To summarize, we've assumed that request validity can be determined based solely on the document on which the request is to be performed, independent of other documents (Section 3.2.3). Other

methods such as `ClicheDb.inTransaction()` should be used when multi-document atomicity is required. Lastly, we also assumed that uniqueness constraints among the documents of a collection only apply to document fields that are immutable (Section 3.2.5 under `updateOne` and `deleteOne`). If we were unexpectedly to need a cliché that requires us to violate the immutability assumption, we may need to also use `ClicheDb.inTransaction()`, or incorporate more checks when determining request validity.

3.3 Subscriptions

There are situations in which one would want the data displayed in an application to be updated in real-time. For example, web apps with a chat feature automatically update the interface to show new messages as they come in. We use the term *reactive* in order to describe cliché actions or web applications that display live-updating data. This section presents how we have enabled Déjà Vu to support reactive cliché actions that can be used in DV-made web applications.

3.3.1 GraphQL

GraphQL[10] is a data query language that cliché actions use to query and mutate cliché data. Cliché authors specify which GraphQL query or mutation is run whenever an action executes or evaluates. GraphQL queries are in turn associated with specific database layer method calls to query or mutate the data in the MongoDB database.

Aside from queries and mutations, GraphQL has a third operation: subscriptions. Executing a GraphQL subscription allows one to subscribe to particular events, such as the creation of or the update to an object. Every time an event happens, a message is sent to those subscribed to that specific event. By incorporating GraphQL subscriptions into the DV architecture, cliché actions can subscribe and be notified about relevant data updates, and subsequently surface those changes to the user interface.

[10]GraphQL: `https://graphql.org/`

3.3.2 Implementation

Using GraphQL subscriptions in DV is not straightforward because of the way DV handles requests. As explained in Section 2.4, requests in DV go through a client-side runtime system that batches any requests that form a transaction, which then sends them to the server-side gateway, which in turn passes them on to the appropriate cliché servers. Below, we describe how we've equipped DV to handle subscriptions.

Communication with WebSockets

Since DV's modules (the runtime system, the gateway, and the cliché servers) previously supported only queries and mutations, they made use of HTTP endpoints for communication. Regular HTTP provides a single response for every request, while subscriptions entail pushing data to clients at any time without the need for a request. While HTTP has support for server-sent events (SSE) which allow servers to push data to clients, the GraphQL libraries available use a different protocol through WebSockets. Both methods are equally capable in terms of enabling servers to push data to clients, so we opted to use the latter as it was simpler to use the existing GraphQL subscriptions library.

The flow of communication using WebSockets between the DV modules is illustrated by Figure 3-6. In the figure, the client-side JavaScript code of a cliché action calls `subscribe()` through the runtime system (1); the runtime system, which maintains an open WebSocket connection to the gateway, attaches an ID and the cliché action information to the subscription request and sends it to the gateway (2); the gateway receives the request, notes the subscription request ID and the associated sender, and forwards the request to the appropriate cliché server (3); the cliché server receives subscription requests and sends notifications each time the relevant event occurs (4). Notifications include the subscription request ID associated with it so that both the gateway and the runtime system would be able to pass on messages received through the WebSocket to the cliché action associated which subscribed to those updates (5, 6).

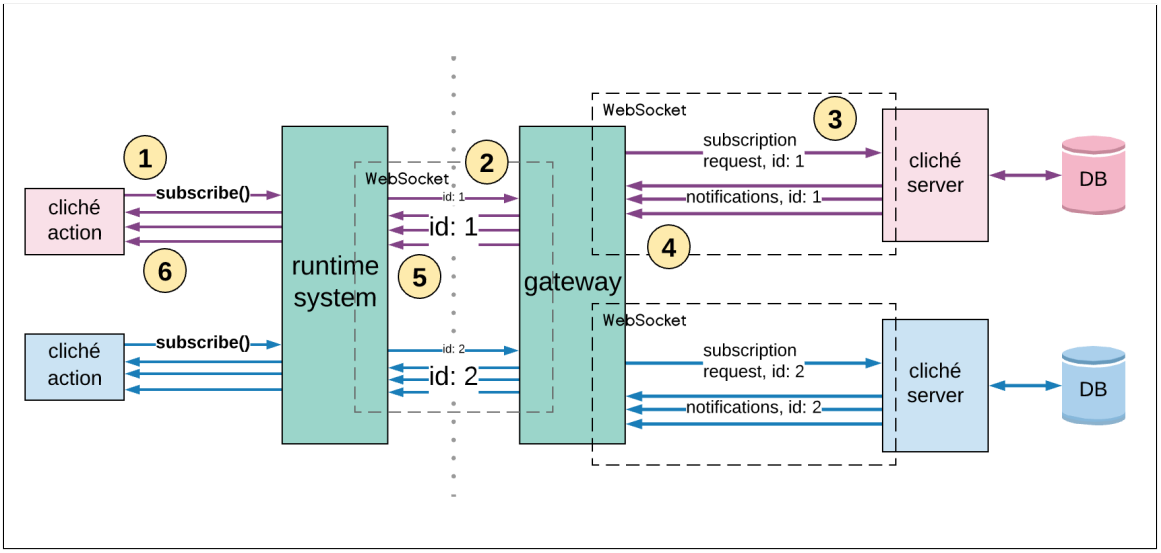


Figure 3-6: WebSocket communication in Déjà Vu

Publish-Subscribe in Cliché Servers

In order for the cliché server to know when to send event notifications to subscribers, cliché authors must specify when to do so. We make use of the publish-subscribe pattern, wherein there are *channels* to categorize each event, and to subscribe to particular type of event, we subscribe to a particular channel. To signal the occurrence of that kind of event, we publish to that channel. A `PubSub` object provided by the GraphQL library we use takes care of processing the publish events and sending them to the relevant subscribers.

As an example, say we want to be notified each time a new comment is written. We would subscribe to the channel `NEW_COMMENT_CHANNEL`, and each time `messages.insertOne(newComment)` is called, cliché authors should include the call `pubsub.publish(NEW_COMMENT_CHANNEL, newComment)` so that subscribers on the channel would get notified. These operations are all internal to the implementation of the cliché and its cliché actions and is hidden from end-users. From their perspective, they will see a reactive list of comments, where new comments automatically show up.

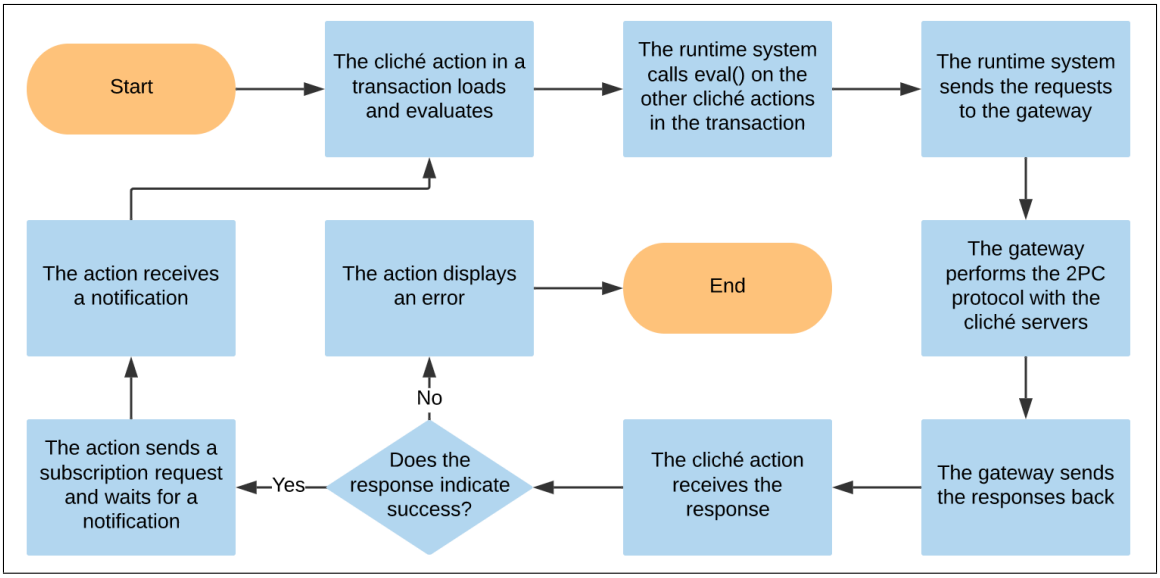


Figure 3-7: A flowchart of the requests a reactive cliché action makes

Securing Subscriptions

In Section 2.3.2, we explained that DV enforces security through transactions that involve actions from the *Authentication* and the *Authorization* clichés. Subscriptions rely on push notifications, where messages are sent from a cliché server to the gateway each time an event occurs. These messages are not tied to any request. Since cliché servers are unaware of DV's security protocol and other clichés, it is not possible for the cliché server to associate the message with other cliché actions such as the *Authentication* cliché's `authenticate` action. The gateway is also not be able to enforce the usual security constraints on messages that are not part of any transaction.

To maintain the use of DV's existing security measures, subscription notification messages do not include any data about the objects in a cliché. They simply serve as a notification to the client-side action that a relevant change has happened. The cliché action should then have code to act appropriately, such as code to fetch the new data. To get the new data, the cliché action should *evaluate* to trigger the evaluation of the other cliché actions in the same transaction, if any. In this manner, if there were any *Authentication* or *Authorization* actions in the same transaction, they would also be evaluated. If the user no longer has access to the data, the transaction would

fail as it should, and the user does not receive data it is not supposed to. Figure 3-7 illustrates the possible sequences of events. The following chapter will discuss the newly-built *Chat* cliché to demonstrate the use of subscriptions.

Chapter 4

Evaluation

In this chapter, we walk through the process of authoring a new cliché, the *Chat* cliché, using the freshly-built cliché-authoring tools and libraries. We also describe our experience in doing so, compare it to past experiences of writing clichés, and from there, report on the successes and shortcomings.

4.1 The Chat Cliché

Whether it's for social, educational, collaborative, or customer support purposes, the chat concept is widely used in web applications. The *Chat* cliché encapsulates the chat concept and allows the exchange of messages in real time among users.

4.1.1 Use Cases and Schema Design

We first identified use cases of the chat concept in web applications. These helped formulate the *Chat* cliché's schema and the cliché actions needed. The use cases are the following:

- A chat should be able to include any number of users. Messages can be exchanged between just a pair of users, but chats should be flexible and allow for more than two users to exchange messages.
- Users should be able to view all the chats of which they are a part.

- Users should be able to view a specific chat. If there are many messages in the chat, then only the most recent messages should be fetched and displayed. Messages in the chat should also be updated in real time. When a user sends a message, the other users in the chat should see the new message without having to refresh the page that displays the chat.

- Users should be able to send a new message to a chat. They should also have the ability to subsequently edit and delete the message.

The functionality that the first bullet point describes is itself a concept that the *Group* cliché already represents. The members of a group are the users included in a chat. It is no surprise that a number of web applications also use the term *group chat* to refer the exchange of messages among multiple users. Based on this insight, we concluded that a data entity of the *Chat* cliché could include an ID that referred to a `group` object's ID. The *Group* cliché has an action `show-groups` that takes in a `memberId` which means that the action can show all the groups of which a user is part. By representing the group of users in a chat with a group in the *Group* cliché, we are automatically able to support the use case described by the second bullet point. We would compose or link *Chat* cliché actions with *Group* cliché actions as in Section 2.3.

The third and the fourth bullet points deal with chat messages. The third suggests that messages should have a timestamp and be sorted by that attribute, while the last one implies that the schema design should make it easy to associate a new message with a chat. We considered two possible schemas, shown in Figure 4-1, that could potentially work with the given scenarios.

In schema 1, the MongoDB document type to be stored is a `Chat` object. `Message`s of a `Chat` are found in an array field of `Chat` and is sorted by increasing timestamp. When new a new `Message` is created, it can simply be appended to the array because its creation timestamp is guaranteed to be later than all previously created `Message`s. Sorting of messages by timestamp thus comes for free. However, schema 1 does not scale well when the embedded `Message`s array in the `Chat` document grows large. The primary mutation operation in chat is the creation of messages, and large arrays

```
Chat {
  id: ID!
  messages: [Message!]
}

Message {
  id: ID!
  content: String!
  timestamp: Float!
  authorId: ID!
}
```

(a) *Chat* cliché schema 1

```
Message {
  id: ID!
  content: String!
  timestamp: Float!
  authorId: ID!
  chatId: ID!
}
```

(b) *Chat* cliché schema 2

Figure 4-1: Proposed schemas for the *Chat* cliché

that are modified frequently can lead to performance problems [9].

We explored an alternative, schema 2, which eliminates the array field by making `Message` a MongoDB document and adding a `chatId` field to it so that we would be able to identify the chat wherein the message is contained. We no longer store explicit `Chat` objects as that would be redundant. We could instead use indices on the `chatId` and the `timestamp` fields so we would be able to quickly fetch `Message`s of a particular `chatId`, and get the latest `Message`s by timestamp. The creation of a new `Message` is still straightforward as that would only be a single insert into the database. Schema 2 is clearly the simpler schema that still satisfies all the use cases for which the *Chat* cliché would be used.

4.1.2 Cliché Actions

Based on the scenarios that the *Chat* cliché should support, this is the initial list of actions we came up with:

- `create-message`: send a message in a chat
- `delete-message`: delete a message sent in a chat
- `show-chat`: show a chat with the given ID,
  with its messages arriving in real-time

```
input CreateMessageInput {
  id: ID
  content: String!
  authorId: ID!
  chatId: ID!
}

input UpdateMessageInput {
  id: ID!
  content: String!
  authorId: String!
}

input ChatMessagesInput {
  chatId: ID!
  maxMessageCount: Int
}

input NewChatMessagesInput {
  chatIds: [ID!]
  lastMessageTimestamp: Float!
}

type Query {
  message(id: ID!): Message!

  # Get the last input.maxMessageChat messages that belong to input.chatId,
  # returning the messages sorted by descending timestamp
  chatMessages(input: ChatMessagesInput!): [Message!]

  # Get all the messages after lastMessageTimestamp
  # that belong to chatId sorted by ascending timestamp
  newChatMessages(input: NewChatMessagesInput!): [Message!]
}

type Mutation {
  createMessage(input: CreateMessageInput!): Message!
  updateMessage(input: UpdateMessageInput!): Boolean
  deleteMessage(id: ID!): Boolean
}

type Subscription {
  newChatMessage(chatId: ID!): Boolean!
}
```

Figure 4-2: *Chat* cliché GraphQL inputs, queries, mutations, and subscription

- `show-message`: show a message with the given ID
- `update-message`: edit a previously sent message

4.1.3 Scaffolding with the DV CLI

We now use the DV CLI to scaffold the *Chat* cliché. We run `dv new cliche chat`, which creates initial actions and server-side code for CRUD operations on the `chat` data entity. These are not the actions and the data entity that we wanted. Since the main entity we've chosen for our schema is `Message` instead of `Chat`, in hindsight, it would be extremely helpful if the `dv new cliche` command were more flexible and allowed users to optionally specify a starting entity name. If it were not specified, then the CLI could default to using the cliché's name as the entity name. Nevertheless, the starting code still proved to be useful as it was relatively simple to change instances of `chat` into `message` in the cliché server code.

As for the cliché actions, instead of renaming the provided ones, we were able to utilize the DV CLI to generate the actions needed through a series of `dv new action` commands. One possible improvement to this part of the process would be the ability to specify the actions that we wanted all at once for the CLI to process. This would benefit a development process where the cliché author comes up with most, if not all, cliché actions altogether in the beginning. The current functionality of the DV CLI works better for incremental development where one would add and write cliché actions one at a time.

4.1.4 Database Queries and Mutations

Figure 4-2 shows the GraphQL queries, mutations, and subscription that the *Chat* cliché actions use. Outside of the code that would enable chat message subscriptions, the CRUD operations on `Message` were all direct translations to the corresponding database layer method call, e.g. `messages.findOne({ id })` for the `show-message` action and the `message(id: ID!)` query. These method calls had also been automatically written by the DV CLI.

Live Updates in `show-chat`

The last remaining action is `show-chat`, which is the action that shows a chat's messages in real-time. The `show-chat` action has two queries associated with it, and one subscription. The client-side code of the action first queries for the last `N` messages of the chat, through the `chatMessages(input: ChatMessagesInput!)` query. If the query is successful, the subscription request `newChatMessage(chatId: ID!)` is sent so that the action receives notifications of new messages in the chat. Lastly, every time the action receives a notification, it re-evaluates, this time calling a different query, `newChatMessages(input: NewChatMessagesInput!)`, to fetch only the new message(s) since the last message it has.

MongoDB allows operations such as `sort()` and `limit()` to be applied to the output of `find()`. As such, to implement the `chatMessages` query, we call

```
find({ chatId: input.chatId })
```

and return the result after sorting by descending message timestamp and then calling `limit(input.maxMessageCount)` to obtain the desired number of most recent messages of the chat. We make similar method calls for the `newChatMessages` query, except that we add

```
{ timestamp: { $gt: input.lastMessageTimestamp }}
```

to the filter parameter of `find()`.

Finally, we implement the publish-subscribe pattern in the cliché server. To publish, we modify the code that creates a message so that it is now also calls `pubsub.publish(NEW_MESSAGE_CHANNEL, newMessage)` whenever a new message is inserted into the messages collection. We take care to make sure that we only do so when the context request type is `commit` or `undefined`. To subscribe, we implement the GraphQL subscription `newChatMessage(chatId: ID!)` so that it subscribes to `NEW_MESSAGE_CHANNEL` and sends event notifications if `newMessage.chatId`

is `chatId`.

4.2 Reflection

After the experience of building the *Chat* cliché, we believe that the newly-built CLI and the `cliche-server` module have made the amount of code to be written mostly trivial and the complexity low. In the `server.ts` file of the *Chat* cliché, we wrote 64 out of the 160 lines of code[1], and of the 64 lines, around half were basic subscription-related code. Most of the rest of the lines of code handle the conversion of JavaScript Date objects to UNIX timestamps[2], which is how chat message timestamps are returned. Table 4.1 summarizes the changes that had to be done to the code the DV CLI generated for each *Chat* cliché action. Only the changes to the reactive `show-chat` action were non-trivial.

When writing the previous clichés, we would usually use an older cliché as a reference to copy boilerplate code from thirty or so files, modifying every occurrence of the cliché name and the object names. To add new cliché actions, we would again consult a previously-created cliché action that uses the same CRUD operation for the HTML, client-side JavaScript, and server-side code. The DV CLI now does those tasks automatically and is able to tailor those files for the specific cliché name and action name.

As for code complexity, the most complex parts of writing the new cliché were coming up with the schema design and understanding how to integrate subscriptions into the cliché code. The former is inevitable whenever there are data entities involved. The difficulty of schema design may also vary depending on the concept being implemented. On the other hand, the latter could be simplified by incorporating some operations into the database layer, and having the DV CLI generate starting code that includes an example of a reactive action such as `show-chat`. The database layer provided by the `cliche-server` module is responsible for much of the success in

[1]The lines of code do not include imports, comments, and blank lines.
[2]UNIX time is the number of seconds since the epoch, 00:00:00 Thursday, 1 January 1970.

Chat action	Changes from CLI-generated code
`create-message`	Added `chatId` and `authorId` as action inputs
`delete-message`	No changes
`show-chat`	Changed most of the code to use `show-message` to display each message of the chat (patterned after the *Event* cliché's `show-events` action that uses `show-event` to display each event), and added subscription-related code as described in Section 3.3.2
`show-message`	Added HTML code to display the message's `timestamp`, `chatId`, and `authorId` fields; and action input booleans to indicate whether those fields should be displayed or not
`update-message`	No changes

Table 4.1: Changes made to DV CLI-generated cliché action code for each *Chat* action

cliché code simplification. One of the most difficult parts in our previous experiences of authoring clichés was implementing and ensuring correctness of the 2PC-handling code, which the database layer now abstracts away.

Overall, even with the possible areas for improvement, this thesis has been successful in crafting a simple and straightforward cliché authoring process. In the next chapter, we offer suggestions on how it can be made better and list design considerations for future iterations.

Chapter 5

Future Work and Conclusion

This chapter concludes the thesis by describing future plans for Déjà Vu, potential aspects of the tools and libraries developed in this thesis on which to improve, prospective system design considerations, and a summary of the work presented.

5.1 Déjà Vu Release

In the future, we hope to collect more user feedback to improve the cliché authorship process. Déjà Vu is slated for public release soon, and when that happens we would be able to gather feedback from both users who would try to use DV to create web applications, and developers who would help author clichés. Feedback from the former could affect what the latter should consider and do when writing cliché code. Observing reception from the latter is also vital in gaining insight as to how potential cliché authors would approach the cliché authorship process.

5.2 DV CLI Improvements

As mentioned in our experience building the *Chat* cliché, there are a couple of refinements that can be done to make the DV CLI more powerful. The command `dv new cliche` could be made more flexible in terms of the name of the data entity it creates instead of simply defaulting to name of the cliché. Some commands could

also take in configuration files that contain a list of actions, data entities, or data entity fields that the command should use. Recruiting more cliché authors to collect more experiences would help polish the DV CLI and any other tools immensely.

5.3 Transaction-Handling Alternatives

There are a few alternatives that the database layer could use to handle the 2PC protocol instead of the current mechanism. Currently, concurrent updates are simply rejected when they occur. Cliché actions would have to re-try the request until it works or until new changes invalidate the request. Normally, it should not take too many retries before the situation is resolved. For instance, once the existing concurrent request finishes, the other request could immediately succeed on the next try. However, if we created queue of pending changes to a document instead, we would be able to remove the performance cost of having to send multiple requests before an action succeeds. We've built the database layer so that it is currently a TypeScript interface whose specific implementation details can be swapped out with another. This would allow us to have multiple different implementations that handle 2PC and they can be compared against one another in terms of limitations and performance.

5.4 Simplifying Subscriptions

With the way subscriptions are set up in DV, cliché authors have to explicitly call `publish()` on the relevant channel whenever an event happens, such as when new messages are created. One way to hide the need to publish is to encapsulate those calls into the database layer for common cases when one would want to publish. We could establish default channels for creation and updates to the entities of a cliché, such that there is always a channel to watch for new `foo`s and another one for updates to a specific `foo`. The `insertOne` method of the database layer would then publish to the former channel regardless of whether the cliché has a subscription for it. This way, it would be easier to incorporate subscriptions into existing and new clichés.

Cliché authors would then only have to define the GraphQL subscriptions that the cliché uses, similar to how it defines queries and mutations.

5.5 Performance, Scaling, and Fault Tolerance

Performance, scaling, and fault tolerance were not major considerations in this thesis and in the development of Déjà Vu in general so far. However, the nature of the way DV's modules communicate would necessitate analyses in these aspects. For instance, the use of 2PC should not slow down a web application to the point that it would be unusable. If they do, then we would need to rethink DV's system architecture so that it provides reasonable performance guarantees. There are a number of possible optimizations for 2PC [4, 10] that can be explored if needed. The DV gateway could also be a single point of failure in the system, so DV should have protocols to handle situations such as those in the future.

5.6 Summary and Conclusion

Déjà Vu is a promising web development platform which we have already used to build over a dozen complex web applications. We aim to make it a tool that a wide range of developers could use, and as such, the cliché catalog should either be thorough enough for various needs, or there should be a low barrier to being able to contribute clichés to the catalog. The latter would also help achieve the former.

The main focus of this thesis was to create a straightforward and simple process to author clichés. The tools and libraries developed were the DV CLI, the `cliche-server` module which includes the database layer, and the DV subscription system to enable reactive cliché actions. Based on our experience building the *Chat* cliché, the newly-built tools have successfully made the cliché authorship process simple and uncomplicated. We hope that the work done in this thesis translates into ease in gaining cliché catalog contributors once Déjà Vu is released in the near future.

Appendix A

Cliché Catalog

Cliché	**Purpose**
Allocator	Distribute resources among consumers
Authentication	Verify the user's identity with a username and password
Authorization	Control access to resources
Chat	Exchange messages in real time with other users
Comment	Share reactions to items
Event	Schedule events
Follow	Receive updates from sources
Geolocation	Locate points of interest
Group	Organize members into groups so that they can be handled in aggregate
Label	Label items so that they can be found later
Match	Connect users after attempting to match with each other
Passkey	Verify the 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	Keep track of scores
Task	Keep track of pieces of work to be done
Transfer	Transfer money or items between accounts

Table A.1: Déjà Vu's cliché catalog

Appendix B

Database Layer Methods

The following are the method signatures (in TypeScript notation) that the database layer allows one to call on a collection of MongoDB documents of type `T`:

```
aggregate(pipeline: Object[],
      options?: mongodb.CollectionAggregationOptions):
mongodb.AggregationCursor<T>

createIndex(fieldOrSpec: string | any,
      options?: mongodb.IndexOptions): Promise<string>

countDocuments(query?: Query<T>,
      options?: mongodb.MongoCountPreferences): Promise<number>

deleteMany(context: Context, filter: Query<T>,
      options?: mongodb.CommonOptions): Promise<boolean>

deleteOne(context: Context, filter: Query<T>): Promise<boolean>

find(query?: Query<T>): Promise<T[]>
```

```
findCursor(query?:  Query<T>):  Promise<mongodb.Cursor<T>>

findOne(query:  Query<T>,
       options?:  mongodb.FindOneOptions):  Promise<T>

findOneAndUpdateWithFn(context:  Context, filter:  Query<T>,
       updateFn:  (doc:  T) => Object | undefined,
       options?:  mongodb.ReplaceOneOptions & UpsertOptions,
       validationFn?:  (doc:  T) => any):  Promise<T | undefined>

insertMany(context:  Context, docs:  T[],
       options?:  mongodb.CollectionInsertManyOptions):  Promise<T[]>

insertOne(context:  Context, doc:  T,
       options?:  mongodb.CollectionInsertOneOptions):  Promise<T>

updateMany(context:  Context, filter:  Query<T>, update:  Object,
       options?:  mongodb.UpdateManyOptions):  Promise<boolean>

updateOne(context:  Context, filter:  Query<T>, update:  Object,
       options?:  mongodb.ReplaceOneOptions):  Promise<boolean>
```

Bibliography

[1] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, May 2002.

[2] Daniel Jackson. Towards a theory of conceptual design for software. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 282–296, New York, NY, USA, 2015. ACM.

[3] Daniel Jackson. *Design by Concept: A New Way to Think About Software*. Pleasant Press, Newton, Massachusetts, February 2019. Prepublication Draft.

[4] Butler W. Lampson and David B. Lomet. A new presumed commit optimization for two phase commit. In *Proceedings of the 19th International Conference on Very Large Data Bases*, VLDB '93, pages 630–640, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[5] Leo A. Meyerovich and Ariel S. Rabkin. Empirical analysis of programming language adoption. *SIGPLAN Not.*, 48(10):1–18, October 2013.

[6] MongoDB. *Atomicity and Transactions - MongoDB Manual*. `https://docs.mongodb.com/manual/core/write-operations-atomicity/`.

[7] MongoDB. *Transactions - MongoDB Manual*. `https://docs.mongodb.com/manual/core/transactions/`.

[8] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[9] Eric Rosenbaum. Thinking about arrays in mongodb, Apr 2013. `https://blog.mlab.com/2013/04/thinking-about-arrays-in-mongodb/`.

[10] George Samaras, Kathryn Britton, Andrew Citron, and C. Mohan. Two-phase commit optimizations and tradeoffs in the commercial environment. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 520–529, Washington, DC, USA, 1993. IEEE Computer Society.

[11] Brandon Satrom. Choosing the right javascript framework for your next web application. *Prog./Kendo UI*, pages 1–34, 2018.