

# Declarative Assembly of Web Applications from Predefined Concepts

by

Santiago Perez De Rosso

Ing., Buenos Aires Institute of Technology (2011)  
S.M., Massachusetts Institute of Technology (2015)

Submitted to the Department of Electrical Engineering and Computer  
Science

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2020

© Massachusetts Institute of Technology 2020. All rights reserved.

Author .....  
Department of Electrical Engineering and Computer Science  
January 30, 2020

Certified by.....  
Daniel Jackson  
Professor of Electrical Engineering and Computer Science  
Thesis Supervisor

Accepted by .....  
Leslie A. Kolodziejcki  
Professor of Electrical Engineering and Computer Science  
Chair, Department Committee on Graduate Students



# Declarative Assembly of Web Applications from Predefined Concepts

by

Santiago Perez De Rosso

Submitted to the Department of Electrical Engineering and Computer Science  
on January 30, 2020, in partial fulfillment of the  
requirements for the degree of  
Doctor of Philosophy

## Abstract

This thesis presents a new approach to web application development, in which an application is constructed by configuring and composing concepts drawn from a catalog developed by experts. A concept is a self-contained, reusable unit of behavior that is motivated by a purpose defined in terms of the needs of an end-user. Each concept includes both client- and server-side functionality and exports a collection of components—graphical user interface elements, backed by application logic and database storage.

To build a web application, the developer imports concepts from the catalog, tunes them to fit the needs of the application via configuration variables, and links concept components together to create pages. Components of different concepts may be executed independently or bound together declaratively with dataflows and synchronization. The instantiation, configuration, linking and binding of components is all expressed in a simple template language.

The approach has been implemented in a platform called *Déjà Vu*. We outline and compare our approach to conventional approaches to web application development and we present results from a case study in which we used our platform to replicate a collection of applications previously built by students for a web programming course.

Thesis Supervisor: Daniel Jackson

Title: Professor of Electrical Engineering and Computer Science



## Acknowledgments

This research was funded by the International Design Center, a collaboration between MIT and the Singapore University of Technology and Design.

I would like to thank my advisor Prof. Daniel Jackson, whose expert guidance made this work possible. I am also grateful to Prof. Rob Miller and Prof. Hal Abelson for being part of my thesis committee and for their insightful comments.

Throughout my years at MIT, I had the good fortune to collaborate with many great MIT students. In particular, I'd like to thank Barry A. McNamara III, Czarina Lao, and Maryam Archie who worked on Déjà Vu for their SuperUROP and M.Eng. Barry developed a graphical environment for Déjà Vu, Czarina built libraries and tools make it easy to implement concepts, and Maryam helped implement the concept catalog and our case study applications. Thank you also to the students that contributed to Déjà Vu through MIT's UROP program: Yunyi Zhu, Shinjini Saha, John Parsons, Stacy Ho, Teddy Katz, and Eric Manzi.

I am grateful for the support and the valuable feedback provided by current and past members of the Software Design Group: Geoffrey Litt, Sergio Campos, Matt McCutchen, Aleksandar Milicevic, Joseph Near, Eunsuk Kang, and Jonathan Edwards.

Thank you to my friends and family. Finally, special thanks to my wife Michelle. This research would not have been possible without her love, support, and encouragement. I dedicate this thesis to my daughter Lucy.

THIS PAGE INTENTIONALLY LEFT BLANK

# Contents

<b>1</b>	<b>Introduction</b>	<b>19</b>
1.1	Motivation . . . . .	19
1.2	Thesis Statement . . . . .	21
1.3	Contributions . . . . .	21
1.4	Thesis Outline . . . . .	22
<b>2</b>	<b>Conventional Approaches</b>	<b>23</b>
2.1	General-Purpose Tools . . . . .	23
2.1.1	Client-Side Programming . . . . .	24
2.1.2	Server-Side Programming . . . . .	26
2.1.3	Tierless Programming . . . . .	27
2.1.4	Full-Stack Components . . . . .	28
2.2	Content Management Systems . . . . .	28
2.3	End-User Development Tools . . . . .	29
2.4	Strengths and Limitations . . . . .	30
2.5	Summary . . . . .	31
<b>3</b>	<b>Our Approach</b>	<b>33</b>
3.1	Concepts as Modules . . . . .	33
3.2	Full-Stack Services . . . . .	34
3.3	Declarative Synchronization . . . . .	35
3.4	Identifier Sharing . . . . .	36
3.5	Strengths and Limitations . . . . .	37

3.6	Summary . . . . .	38
<b>4</b>	<b>Comparison with Conventional Approaches</b>	<b>39</b>
4.1	Rapid Inclusion of End-User Functionality . . . . .	40
4.1.1	Example Application: SecretParty . . . . .	40
4.1.2	Discussion . . . . .	43
4.2	Deep Integration of End-User Functionality . . . . .	44
4.2.1	Example Application: TopMovie . . . . .	44
4.2.2	Discussion . . . . .	49
4.3	Implementation of Custom Behavior . . . . .	50
4.3.1	Example Application: FamilyLog . . . . .	50
4.3.2	Discussion . . . . .	56
4.4	Benefits of Concept Modularity . . . . .	56
4.5	Summary . . . . .	60
<b>5</b>	<b>Building Applications with Déjà Vu</b>	<b>63</b>
5.1	Including and Configuring Concepts . . . . .	66
5.1.1	Choosing Concepts . . . . .	66
5.1.2	Including Concepts . . . . .	66
5.1.3	Configuring Concepts . . . . .	68
5.1.4	Other Application Configuration . . . . .	69
5.2	Linking Components . . . . .	69
5.2.1	Including Components . . . . .	72
5.2.2	Synchronizing Components . . . . .	74
5.3	Building Reactive User Interfaces . . . . .	75
5.4	Specifying Security Policies . . . . .	76
5.5	Styling the Application . . . . .	77
5.5.1	Themes . . . . .	79
5.6	Customizing a Concept Implementation . . . . .	79
5.7	Summary . . . . .	80



<b>6</b>	<b>Platform Semantics</b>	<b>81</b>
6.1	Introduction . . . . .	82
6.2	First Iteration: Components . . . . .	83
6.2.1	Definitions . . . . .	83
6.2.2	Concept Component Behavior . . . . .	88
6.2.3	Rules . . . . .	89
6.2.4	Initial Application Instance Configuration . . . . .	89
6.3	Second Iteration: Clients . . . . .	94
6.3.1	Definitions . . . . .	94
6.3.2	Concept Component Behavior . . . . .	95
6.3.3	Rules . . . . .	96
6.4	Third Iteration: Full Semantics . . . . .	99
6.4.1	Definitions . . . . .	100
6.4.2	Concept Component Behavior . . . . .	100
6.4.3	Rules . . . . .	101
6.5	Core Syntax and Translation . . . . .	107
6.6	Other Considerations . . . . .	109
6.7	Summary . . . . .	109
<b>7</b>	<b>Platform Implementation</b>	<b>111</b>
7.1	Client-Side Library . . . . .	112
7.1.1	Event Dispatching . . . . .	113
7.1.2	Client-Server Communication . . . . .	113
7.2	Gateway Server . . . . .	114
7.2.1	Security . . . . .	114
7.2.2	Transactions . . . . .	119
7.3	Compiler . . . . .	120
7.4	Concept Catalog . . . . .	120
7.4.1	Authoring Concepts . . . . .	121
7.4.2	Criteria for Creating Concepts . . . . .	122

7.5	Summary . . . . .	123
<b>8</b>	<b>Case Study</b>	<b>125</b>
8.1	Research Questions . . . . .	125
8.2	Method . . . . .	126
8.3	Study Subjects . . . . .	127
8.3.1	Project Descriptions . . . . .	128
8.4	Study Replicas . . . . .	130
8.5	Modularity Analysis . . . . .	131
8.6	Effort Savings . . . . .	134
8.6.1	Metric Considerations . . . . .	134
8.6.2	Effort Savings Metric . . . . .	141
8.6.3	Adjustment Factor Values . . . . .	143
8.6.4	Results . . . . .	144
8.7	Quality Analysis . . . . .	146
8.7.1	Usability . . . . .	146
8.7.2	Security . . . . .	154
8.7.3	Performance . . . . .	154
8.7.4	Other Quality Attributes . . . . .	155
8.8	Threats to Validity . . . . .	156
8.9	Summary . . . . .	156
<b>9</b>	<b>Related Work</b>	<b>159</b>
9.1	Programming Paradigms . . . . .	159
9.1.1	Object-Oriented Programming . . . . .	159
9.1.2	Subject-Oriented Programming . . . . .	160
9.1.3	Aspect-Oriented Programming . . . . .	160
9.1.4	Feature-Oriented Programming . . . . .	161
9.1.5	Event-Driven Programming . . . . .	161
9.1.6	Postmodern Programming . . . . .	162
9.1.7	Behavioral Programming . . . . .	162

9.2	Architectural Patterns . . . . .	163
9.2.1	Microservices . . . . .	163
9.2.2	Entity-Component-System . . . . .	163
9.3	Web Application Development . . . . .	164
9.3.1	Content Management Systems . . . . .	164
9.3.2	End-User Development Tools . . . . .	165
9.3.3	Web Frameworks . . . . .	165
9.4	Other Related Work . . . . .	166
9.4.1	Design Patterns . . . . .	166
9.4.2	Federated Databases . . . . .	166
9.5	Summary . . . . .	167
<b>10</b>	<b>Discussion</b>	<b>169</b>
10.1	Future Directions . . . . .	169
10.1.1	Platform Improvements . . . . .	169
10.1.2	Easy Concept Authoring . . . . .	171
10.1.3	A Graphical Environment . . . . .	173
10.2	Open Questions . . . . .	178
10.3	Conclusion . . . . .	179

THIS PAGE INTENTIONALLY LEFT BLANK

# List of Figures

1-1	The same rating concept in multiple applications: applied to businesses on Yelp, products on Amazon, hotels on Hotels.com, applications on Apple’s app store, and browser extensions on Firefox . . . . .	20
1-2	An error in the implementation of the shopping cart concept: setting the quantity to zero doesn’t remove the item from the cart (Image from Nielsen Norman group) . . . . .	20
4-1	The configuration file of <i>SecretParty</i> . . . . .	40
4-2	The style sheet file of <i>SecretParty</i> . . . . .	40
4-3	The code for the landing page of <i>SecretParty</i> . . . . .	41
4-4	The code for the party page of <i>SecretParty</i> . . . . .	42
4-5	The configuration file of <i>TopMovie</i> . . . . .	45
4-6	The style sheet file of <i>TopMovie</i> . . . . .	46
4-7	The code for the landing page of <i>TopMovie</i> . . . . .	47
4-8	The code for the top movies page of <i>TopMovie</i> . . . . .	48
4-9	The code for the show movie component of <i>TopMovie</i> . . . . .	49
4-10	The configuration file of <i>FamilyLog</i> . . . . .	51
4-11	The style sheet file of <i>FamilyLog</i> . . . . .	52
4-12	The code for the landing page of <i>FamilyLog</i> . . . . .	53
4-13	The code for the parent home page of <i>FamilyLog</i> . . . . .	54
4-14	The code for the child home page of <i>FamilyLog</i> . . . . .	55
4-15	Two client-side code variants for <code>submit-post</code> : using Angular and React	57

4-16	Two server-side code variants for <code>submit-post</code> : using a monolithic architecture and microservices . . . . .	58
4-17	<code>submit-post</code> component in <i>Déjà Vu</i> . . . . .	58
5-1	Screenshot of the register user page of <i>Slacker News</i> . . . . .	64
5-2	Screenshot of the submit post page of <i>Slacker News</i> . . . . .	64
5-3	Screenshot of the home page of <i>Slacker News</i> . . . . .	65
5-4	Screenshot of the post detail page of <i>Slacker News</i> . . . . .	65
5-5	Screenshots of two components of <i>Scoring</i> . . . . .	66
5-6	The configuration file of <i>Slacker News</i> . . . . .	67
5-7	Excerpt of <i>Slacker News</i> 's <code>submit-post</code> component . . . . .	70
5-8	Excerpt of <i>Slacker News</i> 's <code>show-post</code> component . . . . .	71
5-9	Excerpt of <i>Slacker News</i> 's <code>upvote</code> component . . . . .	75
5-10	Excerpt of <i>Slacker News</i> 's global style sheet file . . . . .	78
6-1	Inference rule for application component instance configurations . . .	89
6-2	Inference rules for application instance configurations with clients . .	96
6-3	Inference rules for application instance configurations with synchronization . . . . .	102
6-4	Core <i>Déjà Vu</i> syntax in Backus-Naur form . . . . .	107
6-5	Semantic functions that translate grammar rules to a set of initial application instance configurations . . . . .	108
6-6	Semantic functions that translate grammar rules to an initial application server state . . . . .	108
7-1	Architecture of <i>Déjà Vu</i> . . . . .	112
7-2	Excerpt of the component tree for route <code>"/</code> of <i>Slacker News</i> . . . . .	115
7-3	Component path check . . . . .	116
7-4	Transaction structure check . . . . .	117
7-5	Input check . . . . .	118

8-1	An error in the implementation of the authentication concept: special symbols are not allowed, which forces the user to choose a less secure password . . . . .	147
8-2	An error in the implementation of the authentication concept: if a user enters a short password, the error message doesn't say the minimum required length . . . . .	148
8-3	An error in the implementation of the passkey concept caused by an internal inconsistency . . . . .	149
8-4	A coupling of actions from two different concepts: user profiles and authentication . . . . .	150
8-5	A potential coupling of two concepts . . . . .	151
8-6	Update profile in the Déjà Vu implementation of <i>Accord</i> . . . . .	153
10-1	Including and configuring concepts . . . . .	174
10-2	Input/Output binding . . . . .	174
10-3	Input/Output hints . . . . .	175
10-4	Preview mode . . . . .	175

THIS PAGE INTENTIONALLY LEFT BLANK



# List of Tables

5.1	Concept catalog . . . . .	68
7.1	Concept implementations in our catalog . . . . .	121
8.1	Student projects we replicated in Déjà Vu . . . . .	127
8.2	Concept usage in sample applications . . . . .	132
8.3	Libraries and frameworks used in the student implementations . . . . .	135
8.4	Libraries providing some concept functionality . . . . .	139
8.5	Déjà Vu effort savings . . . . .	144

THIS PAGE INTENTIONALLY LEFT BLANK

# Chapter 1

## Introduction

### 1.1 Motivation

As a user, you might have noticed the fundamental similarities between the many applications you use on a day-to-day basis. Maybe it was the day you were scrolling through your Facebook news feed and then through your Twitter feed; or giving a 5-star review to a restaurant on Yelp, and then to a book on Amazon. And just as you, as a user, experience the same rating concept in different variants in multiple applications (Fig. 1-1), so the developers of those applications are, for the most part, implementing that concept afresh as if it had never been implemented before.

In each of these cases, the developer may be implementing something slightly different: a rating of a post in one case, and of a user in another. The premise of this thesis, however, is that these are merely instantiations of the same generic concept, and that if this genericity could be captured, application development could be recast as a combination of pre-existing concepts in novel ways. This might then allow applications to be assembled with much less effort, since the core functionality of the individual concepts would not need to be repeatedly rebuilt.

Moreover, the reuse of concepts could help developers build more usable applications. Even for prevalent and seemingly straightforward concepts like that of a shopping cart, experienced developers can miss important details of the concept's behavior—such as the fact that setting the quantity to zero should remove the item

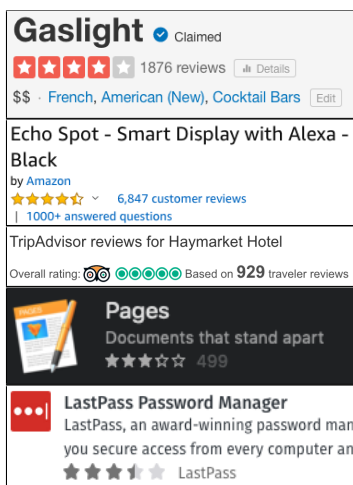


Figure 1-1: The same rating concept in multiple applications: applied to businesses on Yelp, products on Amazon, hotels on Hotels.com, applications on Apple’s app store, and browser extensions on Firefox

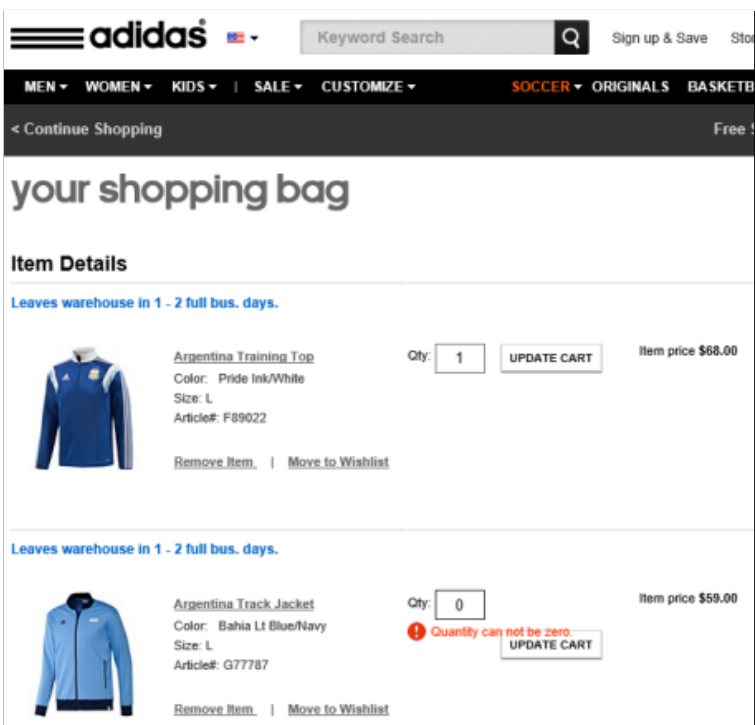


Figure 1-2: An error in the implementation of the shopping cart concept: setting the quantity to zero doesn’t remove the item from the cart (Image from Nielsen Norman group)

from the cart (Fig. 1-2).<sup>1</sup> A catalog of pre-packaged concepts would capture, and let application developers leverage, all the design work done to invent and refine the right concepts.

This thesis presents a new approach to web application development and a new platform, called Déjà Vu, that supports building web applications in this new style. In Déjà Vu, a web application is constructed by combining concepts that are implemented as reusable modules. The assembly requires no procedural code: concept components are glued together by declarative bindings that ensure appropriate synchronization and dataflow. These bindings are expressed in a simple template language, augmenting conventional layout declarations that determine which user interface widgets from which concepts are used, and how they are placed on the page. The net result, as we demonstrate through a case study, is that applications with

<sup>1</sup><https://www.nngroup.com/articles/shopping-cart/>

fairly complex behavior can be built with little effort, and that concept reuse helps prevent several usability problems in the final applications.

## 1.2 Thesis Statement

The thesis of this dissertation is that assembling applications from predefined concepts is a viable approach to developing applications with rich graphical user interfaces and complex behavior. To support this claim, we present results from a case study in which we used our platform to replicate a set of applications previously built by students for a web programming course using standard general-purpose tools. We also compare our approach to conventional approaches to web application development and show, through a set of small examples, how complex application behavior that can be easily implemented using our approach is harder to implement with conventional approaches.

Our hope is that this thesis will promote making concept design and reuse a central aspect of software development and reduce the effort and expertise required to build usable applications with complex behavior.

## 1.3 Contributions

The idea that software should be built from prefabricated parts dates back at least to 1968 [40]. Since then, many mechanisms to support large-scale software reuse have been developed. What’s new in this thesis is what the parts are—implementations of concepts—and how they are put together—by declarative synchronization.

This thesis makes the following contributions:

- A new approach to application development, showing how functionality can be understood as a composition of concepts drawn from a catalog.
- A new platform for application development, that supports independent development of concepts, a simple template language for instantiating and composing concepts, and an infrastructure for executing the resulting applications.

- A case study, in which we used the platform to build a suite of non-trivial sample applications.

## 1.4 Thesis Outline

This thesis begins with an overview of current approaches to web application development in Chapter 2. The core of the thesis is Chapters 3-7, in which we explain the key ideas in our approach, compare our approach to conventional approaches to web application development, show what building applications with Déjà Vu looks like, and discuss the semantics and implementation of our platform. Chapter 8 presents the case study we conducted. Finally, we discuss related work in Chapter 9 and conclude in Chapter 10 with a discussion on future directions and open questions.

An initial report on the work described in this thesis appears in [51]. Compared to [51], this thesis includes a set of small examples to illustrate the benefits of our approach compared to conventional approaches to web application development, formalizes the semantics of our platform, and provides a much fuller analysis of the applications we replicated in our case study.

# Chapter 2

## Conventional Approaches

The purpose of web development is to build a web application. A web application is a type of distributed application that consists of a server, a database, and many clients. The client code runs on a web browser and communicates with the server over the internet or over some other computer network. The server interacts with the database and provides functionality for clients.

In this chapter, we give an overview of three popular approaches for building web applications: using general-purpose tools (§2.1), using a content management system (§2.2), and using an end-user development tool (§2.3). Finally, we conclude the chapter with a discussion on the strengths and limitations of these approaches (§2.4).

### 2.1 General-Purpose Tools

Using general-purpose tools, a developer writes the code that runs on clients and the code that runs on the server. Client-side or front-end programming (§2.1.1) is concerned with the development of the client-side code of the application, which deals with the user interface of the application. Server-side or back-end programming (§2.1.2) is concerned with the development of the server-side code of the application, which implements business logic, retrieves and updates data from the database, and serves views for clients. A tierless programming language (§2.1.3) blurs the traditional

client-server distinction and allows a developer to develop both client- and server-side functionality using one programming language. A full-stack component (§2.1.4) is a graphical user interface element that interacts with a web service and allows a developer to include both client- and server-side functionality into the application at once. These components are said to be full-stack because they encapsulate both client- and server-side functionality.

### 2.1.1 Client-Side Programming

The client-side functionality of a web application is typically developed using HTML, CSS, and JavaScript. HTML is a markup language for defining the meaning and structure of web content. HTML uses markup to annotate content, such as text or images, for display in a web browser.

CSS is a style sheet language for describing the presentation of an HTML document. A style sheet consists of a set of rules, where each rule has a set of selectors and a list of property-value declarations. The selectors declare which HTML elements the rule should apply to, and the list of property-value declarations specify how the matching HTML elements should be styled. For example, a developer can write a selector to match all heading elements, and a property-value declaration to change the font color of the matched elements to blue. The result of writing this rule is that when a browser renders the HTML document, all headings will appear in blue.

JavaScript is a scripting language that allows a developer to build interactive web pages. With JavaScript, a developer can, for example, attach an event handler to an HTML element and in response to some user interaction, such as a button click, mutate the contents of the web page or send a request to the server.

Nowadays developers have more language options for client-side programming. For example, TypeScript<sup>1</sup> is a popular<sup>2</sup> alternative to JavaScript that adds optional static typing to the language, and SASS<sup>3</sup> extends CSS with features such as variables and

---

<sup>1</sup><https://www.typescriptlang.org/>

<sup>2</sup><https://2018.stateofjs.com/javascript-flavors/overview/>

<sup>3</sup><https://sass-lang.com/>



inheritance. In the end, however, all these other languages end up being compiled to HTML, CSS, and JavaScript. This is because web browsers can only execute JavaScript and render HTML documents styled with CSS.

## Client-Side Libraries and Frameworks

A modern web application is typically developed using a front-end framework or library. The three most popular<sup>4</sup> client-side libraries and frameworks are React<sup>5</sup>, Angular<sup>6</sup> and Vue<sup>7</sup>. While these frameworks have differences in terms of functionality, they all provide a way for the developer to declare components and data bindings.

A component is an independent, reusable, graphical user interface element. A component can have input properties and an internal state. A component can also produce outputs for other components to use. Components group together HTML, CSS and JavaScript code, offering a different kind of modularity to the conventional modularity in which, for example, there's one HTML file per web page and one CSS and JavaScript file for the entire application. The benefit of components is that they allow a developer to split the user interface of an application into reusable pieces, and think about each piece in isolation from the rest of the functionality.

All three frameworks also have some built-in mechanism to declare data bindings. A data binding establishes a connection between a data source and data consumer. In a data binding, each data change is propagated to the consumer automatically. This feature could be used in, for example, an HTML template to display the value of a JavaScript variable and have the framework automatically refresh the contents of the component every time the value of the variable changes.

These two features, components and data bindings, simplify client-side programming because they give the developer a way to capture and reuse graphical user interface elements and saves the developer from having to write custom code to propagate data updates.

---

<sup>4</sup><https://2018.stateofjs.com/front-end-frameworks/overview/>

<sup>5</sup><https://reactjs.org/>

<sup>6</sup><https://angular.io/>

<sup>7</sup><https://vuejs.org/>

## 2.1.2 Server-Side Programming

Server-side code can be written in any number of programming languages. The most popular languages for server-side programming include PHP, Java, C#, JavaScript, Ruby, and Python.<sup>8</sup> These are all very powerful, general-purpose programming languages that give the developer the flexibility to implement complex business logic. However, to implement server-side functionality, it is not enough for a developer to know how to program in one of these languages. A developer must also have some familiarity with the HTTP protocol, API design and database query languages, so as to be able to process client requests and interact with the database.

For example, it is up to the developer to choose the best database model for the application. There are many types of databases: relational, document-oriented, graph, and so on. Each type has its own query language and different performance trade-offs. Similarly, there are different models for building an application programming interface (API). An API is a communication protocol between a client and a server. An API establishes the type of requests a client can make to the server, and the type of responses the client can expect to receive from the server. The two most popular models for API design and implementation are Representational State Transfer (REST) [16] and GraphQL<sup>9</sup>. Each model has its own approach to declaring and implementing a server API and different trade-offs.

For transmitting data between a client and the server, web applications typically use JSON. JSON is a language-independent data interchange format that originated as the object literal language of JavaScript. JSON uses human-readable text to transmit data objects consisting of attribute-value pairs. JSON is also widely used as a configuration language. For example, NPM<sup>10</sup>, a package manager for JavaScript, expects package authors to create a JSON file to specify the name of the package, its software dependencies, and other relevant package information.

---

<sup>8</sup>[https://w3techs.com/technologies/overview/programming\\_language](https://w3techs.com/technologies/overview/programming_language)

<sup>9</sup><https://graphql.org/>

<sup>10</sup><https://www.npmjs.com/>

## Monolithic and Microservices Architecture

The server-side functionality of an application can be implemented as one program, or it can be split into multiple independent programs that implement different aspects of the server-side functionality. In a monolithic architecture, the server-side functionality of an application is implemented as one program. In a microservices architecture [44], the server-side functionality of an application is split into a set of small programs, called microservices, that communicate with each other via HTTP.

Compared to a monolithic architecture, a microservices architecture enforces strong module boundaries between server-side functionality, allows different parts of the server-side functionality to be implemented using different programming languages and storage technologies, and allows each service to be tested and deployed independently. But this flexibility comes at a cost: the developer now has to additionally write code to integrate different remote services, which is not necessary in a monolithic architecture because all the functionality is part of the same program. Service integration code can be very complex since remote calls can fail and achieving strong data consistency might require the developer to implement complex communication protocols between the different services.

### 2.1.3 Tierless Programming

A tierless programming language, such as Links [10] and Ur/Web [8], combines within one language all the pieces of web application development: client- and server-side programming plus database querying. For example, instead of having to define a server-side API and write code to send HTTP requests from the client, a developer can invoke a server-side function from client-side code as it were any other client-side function.

Compared to a general-purpose language, a tierless programming language can reduce some of the complexity in web application development and help prevent several kinds of programming errors that can lead to security problems or bugs in web applications. But a tierless programming language, like a general-purpose programming

language, targets professional developers and still requires the developer to write most of the end-user behavior of the application.

### 2.1.4 Full-Stack Components

Some web API services offer full-stack components. These components are said to be full-stack because they include both client- and server-side functionality: full-stack components can make requests to a web API service without requiring the developer to write code to mediate such communication. However, deeply integrating the functionality provided by a full-stack component with the rest of the application functionality might require the developer to open the full-stack black box and write server-side code.

For example, a developer can use Disqus<sup>11</sup> to add commenting functionality to an application. If the commenting functionality is isolated from the rest of the application functionality, a developer can add commenting functionality to the application by only including some JavaScript and writing a little HTML. But if the developer wants to integrate commenting with other application functionality such as upvoting for example, then the developer would have to write server-side code. For example, the developer might have to create a notification hook for the server of the application to be notified by the web API service when a new comment is created, so that the new comment can be given an initial score in the application database.

## 2.2 Content Management Systems

Another option for web application development is to use a content management system, such as Drupal<sup>12</sup> or WordPress<sup>13</sup>. A content management system supports the creation of websites, such as blogs and media sites, whose functionality primarily involves reading and writing content. In many content management systems, the content management functionality is very sophisticated and can include features such

---

<sup>11</sup><https://disqus.com/>

<sup>12</sup><https://www.drupal.org>

<sup>13</sup><https://wordpress.com/>

as version control, access control, workflow and publishing control, and advanced content templates.

Many content management systems provide a large suite of plug-ins that allow a website to be extended with new features. These plug-ins are full-stack, and importing and configuring them is usually straightforward. But because plug-ins lack a generic composition mechanism, they can usually only be combined in certain predefined ways. More application-specific combinations typically require modifying server-side or content-management-system code. For example, let's say the developer wants to let users comment on blog posts and upvote comments by other users. The developer might be able to include a plug-in to add comments to the application, but if the commenting plug-in is not designed to work with the plug-in providing the upvoting functionality, the developer would typically have to write complex glue code to integrate the two plug-ins.

## 2.3 End-User Development Tools

A third option is to use an end-user development tool. An end-user development tool enables people who are not professional software developers to create software artifacts [37]. Some examples of end-user development tools include low-code development platforms [57], such as OutSystems<sup>14</sup>, Mendix<sup>15</sup>, and Microsoft Power Apps<sup>16</sup>, and the many tools and languages for programming data-centric applications, such as Mavo [64] and Espalier [39]. These tools typically offer a visual interface and domain-specific languages to specify a schema, queries, updates, and views. For standard CRUD (create-read-update-delete) functionality, these platforms can work well, and they allow arbitrary customization of queries and updates. Code for handling the mechanics of graphical user interface elements, data storage, server requests, etc., is provided and hidden, and this is a great help. But an end-user development tool still requires the developer to write all the code for the application logic. Moreover, func-

---

<sup>14</sup><https://www.outsystems.com>

<sup>15</sup><https://www.mendix.com>

<sup>16</sup><https://powerapps.microsoft.com>

tionality that is not expressible in the language—because it involves some algorithmic complexity, or a more complex user interface widget—must be provided by pre-built plug-ins. As with content management systems, such plug-ins can only be composed in ad hoc ways, and may require complex glue code.

## 2.4 Strengths and Limitations

In summary, these are the strengths and limitations of conventional approaches to web application development:

- *General-Purpose Tools.* General-purpose tools are flexible, but require significant programming expertise. While any web application can be developed using this approach, the developer needs to know how to write complex code that deals with HTTP requests and database queries. Full-stack components can help incorporate client- and server-side functionality quickly, but only if the full-stack component behavior can be kept largely isolated from the rest of the application functionality. Deeply integrating the full-stack functionality with the rest of the application functionality typically requires writing server-side code to get notified when certain events occur, so that other parts of the application can be updated accordingly.
- *Content Management Systems.* The built-in support that content management systems provide for managing the creation and modification of content allows a user with no programming experience to easily add, modify and remove web content. Moreover, many content management systems have a large suite of plug-ins, which allows a developer to easily add full-stack behavior. A problem, however, is that plug-ins lack a generic composition mechanism. Therefore, integrating different plug-ins may require writing complicated server-side code.
- *End-User Development Tools.* While typically not as flexible as general-purpose tools, end-user development tools allow complex behavior to be specified at a very high level. A problem, however, is that the developer still has to write most

of the end-user behavior of the application. The built-in functionality is usually limited to graphical user interface elements such as text boxes and buttons. It is still up to the developer to write code to specify what would happen when, for example, the end-user clicks on a button. Plug-ins that allow end-user behavior to be quickly added to an application lack a composition mechanism, and combining them with other plug-ins can be as difficult as combining plug-ins in a content management system.

## 2.5 Summary

There are three main approaches to web application development: using general-purpose tools, a content management system, or an end-user development tool. General-purpose tools are flexible, but require significant programming expertise. A developer needs to know how to handle HTTP requests, how to design and implement APIs, write database queries, and implement client-side interactive functionality. In a content management system a developer can quickly add full-stack functionality to an application through plug-ins. But plug-ins lack a generic composition mechanism, so deeply integrating different plug-ins can be challenging and require writing server-side code. An end-user development tool typically offers a visual interface and domain-specific languages for specifying a schema, queries, and updates. Code for handling data storage, server requests, and so on is provided and hidden. But the developer still has to write most of the end-user behavior of the application.

In the next chapter, we present our approach to web application development, which allows the developer to include full-stack functionality to the application and compose the functionality declaratively, without writing any server-side code. Our approach addresses many of the limitations of the conventional approaches to web application development that we discussed in this chapter.

THIS PAGE INTENTIONALLY LEFT BLANK



# Chapter 3

## Our Approach

There are four key ideas in our approach to web application development. The first two ideas, concepts as modules (§3.1) and full-stack services (§3.2), define the elements Déjà Vu applications are made of: full-stack implementations of concepts. The last two ideas, declarative synchronization (§3.3) and sharing identifiers (§3.4), determine the mechanism by which concepts are combined to achieve some greater purpose: through the synchronization of server-side actions and the binding of concept entities by shared identifiers.

### 3.1 Concepts as Modules

In our approach, concepts are the building blocks of applications. A concept is a self-contained, reusable increment of functionality that is motivated by a purpose defined in terms of the needs of an end-user [28]. For example, a posting concept manages the creation and display of posts, and a scoring concept lets users assign scores to items and display items sorted by total score value.

A concept is a new type of module that encapsulates data and behavior that implements one end-user purpose. A concept is more fine-grained than a traditional plug-in in a content management system and it is also more fine-grained than a microservice. For example, a plug-in or microservice would generally treat the concepts associated with user posting, such as scoring and comments, as part of a single plug-

in or microservice. Also, a concept is more coarse than an object in object-oriented programming or an abstract data type [38]. For example, implementing a transfer concept might require multiple objects or abstract data types: an account type to save the balance of an account and a transfer type to record a transfer between accounts.

By making concepts the units from which Déjà Vu applications are made of, our approach emphasizes the design and implementation of concepts, making concepts and their composition a central aspect of application development.

## 3.2 Full-Stack Services

A concept encapsulates all the code necessary to support its end-user purpose and exports a set of components. All the front-end and back-end code that implements the concept functionality is hidden from the developer, who only needs to be concerned with the exported components of a concept. Each component of a concept consists of a front-end widget with input and output properties and a set of associated server-side actions.

A Déjà Vu application instance includes a set of concept instances that run in parallel. A concept instance is a concrete occurrence of a concept that exists only during the runtime of a Déjà Vu application. For example, if we include a scoring concept in a Déjà Vu application, a new instance of the scoring concept is created each time we run the application and terminated once we stop the application. The data of a concept instance, however, could persist after the concept instance is terminated if the developer chooses so. Of course, a Déjà Vu application can be deployed like a normal web application, and an instance of scoring will be deployed as part of the application as well. Concept instances are not multi-tenant: each Déjà Vu application has its own set of concept instances, even if they share the same concepts.

Each concept instance runs independently of other concept instances: concept instances cannot communicate with each other or share any data. Abstractly, a concept instance is a state machine that changes state in response to actions issued by the end-user of the application through the front-end, or in response to system events,

such as the loading of a component into the user’s web browser. Since each concept instance includes its own components, server, and database, a concept instance is a full-stack service in its own right: it provides a service to the end-users of a *Déjà Vu* application that includes client- and server-side functionality and data persistence.

### 3.3 Declarative Synchronization

Let’s say we want to create an application in which end-users can create posts with an initial score. To do this, we include a posting and scoring concept to our application, and we create an application component that includes the create post component of the posting concept and the create score component of the scoring concept. So far, we have an application that enables end-users to create new posts by interacting with the create post component and to create new scores by interacting with the create score component.

However, our goal is for an end-user to be able to create a post with an initial score at the same time. For this, we need the server-side action of the create post component and the server-side action of the create score component to occur together. In *Déjà Vu*, to make the server-side actions of components occur together, the developer indicates, in the same file that specifies the included components, which actions of the components are to be synchronized. When a component runs its server-side action, *Déjà Vu* will run the server-side actions of all the synchronized components. Thus, if in our application we synchronize the create post component with the create score score component, when an end-user interacts with create post to create a new post, both a post and a score object will be created at the same time.

The synchronization of actions is transactional, ensuring that all or none of the concept server-side actions that are part of a transaction occur. That is, if one server-side action fails, all the other server-side actions in the transaction will fail as well. A failed transaction produces no side-effect on the state of the concept instances involved in the transaction—it is as if the transaction never occurred in the first place.

There are various reasons why a concept instance might choose to fail the execution of a server-side action. Perhaps the input values given to the action are invalid, or perhaps the purpose of the action is to perform a server-side check. For example, the server-side action of the create score component of the scoring concept fails the transaction if the given score value is negative. The same scoring concept could also have an action that verifies that a given target has a score, and fail the transaction if it doesn't. A developer can include the component associated with this check action in a transaction with other concept components to, for example, only let a user create a new score if the target of the score doesn't have a score already.

The same transaction mechanism can be used for security. Concepts are provided for user authentication and access control whose actions are designed to fail when access is denied. By synchronizing the authentication components with other concept components, a developer can ensure that certain actions only occur when the logged-in user has the permissions required to perform the action.

Of course, not all components need to be synchronized. Components can also be placed on a web page so that they run independently of one another. As an end-user interacts with the components on the page, the server-side actions of different concept instances can be triggered by the end-user, and run independently from one another. The inputs and outputs of components can still be related however, even if the components are not synchronized.

### **3.4 Identifier Sharing**

Often, concept instances hold distinct views of entities that can be thought of as shared. For example, when a scoring concept is combined with a posting concept, one can imagine a single entity having both a scoring view and a posting view: the scoring view contains the scoring value of the entity and the posting view contains the textual post. But, as explained above, there is no explicit sharing of data between concepts. Instead, common identifiers are used to implicitly bind distinct entity views. This is achieved using a special platform function that generates an identifier that

is, for practical purposes, distinct from any other identifier ever generated by the platform function—the identifier is a so-called universally unique identifier (UUID) [36]. The identifier is then passed as input to the two components, ensuring that the entities they create will subsequently be associated by the shared identifier.

By convention, concept components for creating entities take an identifier as input. When an identifier value is given as input, the component uses the given identifier as the identifier of the new entity to create, instead of choosing a fresh identifier server-side. In those cases in which the entities being created can refer to other entities, the component has multiple inputs that take identifier values. For example, the create score component has an input for the identifier of the score entity being created, one for the identifier of the source entity giving the score, and another input for the identifier of the target entity that is being assigned the new score.

### 3.5 Strengths and Limitations

Our approach, like in content management systems, allows the developer to quickly include full-stack functionality at once by adding a new concept to the application. Unlike plug-ins in content management systems, however, our concepts have a generic composition mechanism that allows a developer to compose different concepts by synchronizing their actions, without requiring the developer to write any server-side code. Since concept components include client-side interactive behavior, there’s no need for a developer to write code to respond to user events or send HTTP requests. A developer has only to specify what actions should be synchronized, and the platform will take care of running the server-side actions in a transaction.

A limitation of our approach is that a developer is restricted to building what can be built with the concepts available in the catalog. While a developer can add new concepts to the catalog, concepts are implemented using general-purpose tools and thus require more expertise and effort. However, as we’ll see later, our composition mechanism is flexible enough that application-specific behavior that might appear

like it would need a new concept can often be implemented by combining existing concepts.

## 3.6 Summary

In our approach, concepts encapsulate full-stack functionality that implement an end-user purpose. Concepts can be combined together by synchronizing server-side actions. To link different concepts entities the developer can use a built-in component to generate a unique identifier and share the identifier with different concept components, so that the components use the same identifier when they create a new entity.

Our approach allows the developer to quickly include full-stack functionality to an application. The mechanism by which concepts are put together allows a developer to integrate concepts, declaratively, without writing complicated glue code.

In the next chapter, we are going to present three example applications that illustrate the benefits of our approach compared to the conventional approaches to web application development we discussed in the previous chapter.

# Chapter 4

## Comparison with Conventional Approaches

In this chapter, we present three small example applications built using our Déjà Vu platform. The goal is to show, by example, the key benefits of Déjà Vu compared to conventional approaches. The goal is not to thoroughly explain how to build applications using Déjà Vu—Chapter 5 will provide that explanation.

The three small example applications are *SecretParty*, *TopMovie*, and *FamilyLog*. *SecretParty* shows how easy it is to rapidly include end-user functionality to a Déjà Vu application (§4.1). *TopMovie* shows how it is possible to deeply integrate different concepts in Déjà Vu, without having to write any server-side code (§4.2). Finally, *FamilyLog* shows how custom behavior that might at first appear like it would need a very specific concept can be implemented by combining existing concepts from the catalog (§4.3). To conclude the chapter, we include a discussion on the benefits of our concept modularity compared to the conventional modularity mechanisms and software architectures used in general-purpose tools (§4.4).

```

1 {
2   "name": "secretparty",
3   "usedConcepts": {
4     "passkey": {},
5     "event": {},
6     "geolocation": {},
7     "chat": {}
8   },
9   "routes": [
10    { "path": "", "component": "landing" },
11    { "path": "/party", "component": "party" }
12  ]
13 }

```

Figure 4-1: The configuration file of *SecretParty*

```

1 @import "~@deja-vu/themes/scss/minimalist.scss";
2 main { margin: 30px; }
3 .side-by-side { display: flex; }
4 .map { height: 300px !important; }
5 .evt { margin-right: 20px; }

```

Figure 4-2: The style sheet file of *SecretParty*

## 4.1 Rapid Inclusion of End-User Functionality

### 4.1.1 Example Application: SecretParty

In *SecretParty* users can create events with a secret code, date, time, and location. Users that know the secret code of a party can input the code to see the party date, time, and location. Users can also chat with guests to share any information or ask any questions about the party.

The entire code for the Déjà Vu application is shown in Figs. 4-1 through 4-4, together with screenshots of how the application appears to end-users. The code for the application consists of a JSON configuration file to include concepts and configure other application information (Fig. 4-1), a SASS style sheet file to customize the presentation of the application (Fig. 4-2) and an HTML file for each component



```

1 <dv.component name="landing">
2   <main>
3     <h1>View Party</h1>
4     <dv.tx>
5       <passkey.sign-in /><dv.link href="/party" />
6     </dv.tx>
7     <h1>New Party</h1>
8     <dv.tx>
9       <dv.gen-id /><dv.status />
10      <passkey.create-passkey id=dv.gen-id.id hidden=true />
11      <event.create-event class="evt" id=dv.gen-id.id
12        showOptionToSubmit=false />
13      <div class="map">
14        <geolocation.create-marker-from-map id=dv.gen-id.id
15          showOptionToSubmit=false />
16      </div>
17      <dv.button-link href="/party">Create Party</dv.button-link>
18    </dv.tx>
19  </main>
20 </dv.component>

```

# View Party


Passkey Code \*

---

Validate

# New Party

Starts On \*

02/07/2020 


---

Start Time \*

05:00 PM

---

Ends On \*

02/07/2020 

---

End Time \*

10:00 PM

---

[Create Party](#)

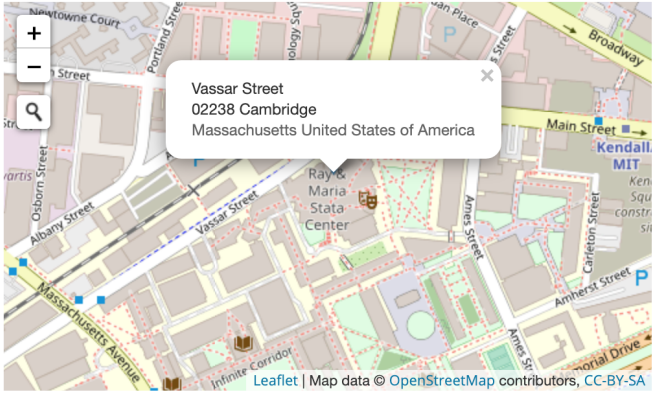


Figure 4-3: The code for the landing page of *SecretParty*

```

1 <dv.component name="party">
2   <main>
3     <passkey.logged-in />
4     <h1>{{passkey.logged-in.passkey?.code}}</h1>
5     <event.show-event id=passkey.logged-in.passkey?.id waitOn=['id'] />
6     <geolocation.show-marker id=passkey.logged-in.passkey?.id
7       hidden=true />
8     <div class="map">
9       <geolocation.display-map expectMarkers=true
10        markers=[geolocation.show-marker.loadedMarker] />
11     </div>
12     <h2>Chat</h2>
13     <chat.show-chat id=passkey.logged-in.passkey?.id
14       showId=false showMessageId=false />
15     <chat.create-message authorId="anonymous"
16       chatId=passkey.logged-in.passkey?.id />
17   </main>
18 </dv.component>

```

alamo

Fri Feb 7, 2020 5:00 PM - 10:00 PM

Chat

should we bring food?  
 Tue Dec 10 2019 15:20:25 GMT-0800  
 anonymous

Content \*

Create Message

Figure 4-4: The code for the party page of *SecretParty*

(Fig. 4-3 and Fig. 4-4). The application *SecretParty* has two pages: a landing page that lets users create a new party or input a party code to view an existing party (Fig. 4-3), and a party page that shows the party information and lets users chat (Fig. 4-4). The JSON configuration file (Fig. 4-1) specifies the concepts we use in the application (lines 4-7) and the routes for the two pages (lines 10-11). We include *Passkey* (line 4) to implement the secret code behavior, *Event* and *Geolocation* (lines 5-6) for the party date and location respectively, and *Chat* (line 7) to create a chat room for party guests.

To create a party (Fig. 4-3), we synchronize, using the `dv.tx` tag, the create components of passkey (line 10), event (lines 11-12) and geolocation (lines 14-15). The effect of doing so is that when the user clicks on the “Create Party” button, a new passkey, event, and geolocation marker is created. The passkey code is generated server-side by the action of `create-passkey`. To bind the different entities together, we include the built-in component `dv.gen-id` (line 9) and share the generated identifier with all the create components (lines 10, 11, and 14). To redirect the user to the party page after the party is created we include a `dv.button-link` as part of the transaction (line 17). The button link will trigger the transaction when the end-user clicks on the button and it will redirect the end-user to the party page if the transaction succeeds. Finally, in the party page code (Fig. 4-4), we retrieve the party identifier from the logged-in passkey (line 3), and use it to show the secret code (line 4), date (line 5), location (line 6), and chat room (line 13) of the party.

In the application style sheet file (Fig. 4-2), we include one of the built-in themes of our platform (line 1) and define some CSS rules for styling (lines 2-5).

### 4.1.2 Discussion

Note how with less than 60 lines of HTML, SASS and JSON code, and no JavaScript or server-side code, we are able to implement an application that has quite a lot of rich behavior: passkey generation and verification, maps, events, and chat functionality. Simply by writing a little JSON and HTML we can include concepts, such as *Chat* and *Geolocation*, that package a lot of end-user functionality.

Implementing *SecretParty* using general-purpose tools would require writing complicated client- and server-side code. For example, while there are many APIs that let you include a map to your application, we would still have to write JavaScript to retrieve the location the user selected on the map, and include it in the new-party HTTP request so that we can save the location in the database. The server-side code would have to save the party information in the database and retrieve the information on the show-party page. The server-side code would also have to authenticate user-provided passkey codes. On the bright side, it might be possible to use a full-stack component for chat functionality, since the chat functionality in *SecretParty* is relatively isolated from the rest of the application functionality.

An end-user tool would require us to write all of the end-user behavior of *SecretParty*. For example, we would have to specify what happens when the user clicks on the create party button: obtain data from the form and save it to the database.

In a content management system we could define a party content type, which can even include map information through a map plug-in. Through some advanced configuration we could open party creation to any user of the application and not just the administrator of the application. But unfortunately an end-user would still be forced to create an account before creating a party. This means that an anonymous user would not be able to create a secret party without registering.

## 4.2 Deep Integration of End-User Functionality

### 4.2.1 Example Application: TopMovie

In *TopMovie*, users can post movie titles and upvote movies submitted by other users. The movies with the most votes appear towards the top of the top movies page. The code for *TopMovie* is shown in Figs. 4-5 through 4-9, together with screenshots of how the application appears to end-users. The application has two pages: a landing page for users to register or sign-in (Fig. 4-7) and the top movies page that shows all

```

1 {
2   "name": "topmovie",
3   "usedConcepts": {
4     "movie": {
5       "name": "property",
6       "config": {
7         "schema": {
8           "title": "Movie", "type": "object",
9           "properties": { "title": { "type": "string" } },
10          "required": ["title"]
11        }
12      }
13    },
14    "authentication": {},
15    "scoring": {}
16  },
17  "routes": [
18    { "path": "", "component": "landing" },
19    { "path": "/top", "component": "top-movies" }
20  ]
21 }

```

Figure 4-5: The configuration file of *TopMovie*

movies sorted by votes (Fig. 4-8). We also define a `show-movie` component that is used in the top movies page to display each movie (Fig. 4-9).

For posting movie titles we use the *Property* concept. *Property* is a concept that provides simple CRUD behavior of objects. The schema of the objects the concept will be dealing with can be configured. In *TopMovie* we include *Property* and configure it to save objects with a title field, which represents the movie title (Fig. 4-5, lines 7-11). In addition to *Property*, we include *Authentication* (line 14) to register users and *Scoring* (line 15) for keeping track of movie votes.

The application *TopMovie* requires the deep integration of three separate concepts: authentication, posting and scoring. We want the same users that post movies to be the ones than can upvote movies, and we want to prevent a user from upvoting a movie they posted. We achieve this by sharing identifiers and synchronizing actions. In the submit movie form that appears on the top movies page (Fig. 4-8), we wrap with the `dv.tx` tag the `create-object` component of *Property* (line 7), the `create-score`

```

1 @import "~@deja-vu/themes/scss/sugar.scss";
2
3 .side-by-side {
4   display: flex;
5   justify-content: space-evenly;
6 }
7 .lr-margin {
8   margin-left: 20px;
9   margin-right: 20px;
10 }
11 nav h1 {
12   text-align: center;
13   font-size: 40px;
14 }
15
16 .movie { display: flex; }
17 .movie-info { padding-top: 9px; }

```

Figure 4-6: The style sheet file of *TopMovie*

component of *Scoring* (line 8), and the `authenticate` component of *Authentication* (line 10). The `create-object` component creates a movie object, `create-score` initializes the movie score, and `authenticate` authenticates the logged-in user. The *Scoring* concept, unless it is configured otherwise, allows a source to score a target only once. By creating a score with the source ID set to the logged-in user (line 9) we prevent a user from upvoting a movie they submitted.

Deeply integrating these concepts also requires creating a unified view that includes the movie title and number of points. Concept components are higher-order: they can take components as input. It is standard convention for concept components that may show multiple entities to have a component input to allow the developer to customize the way each entity is shown. For example, the component `show-targets-by-score` of *Scoring* has a `showTarget` input. If the developer specifies a value for `showTarget`, the `show-targets-by-score` component will use the provided component to show each target, instead of using a default `show` component.

On the top movies page, we bind the `showTarget` input of `show-targets-by-score` to the `show-movie` component (lines 16-17). The `show-movie` component (Fig. 4-9)

```

1 <dv.component name="landing">
2   <nav><h1>topmovie</h1></nav>
3   <main class="side-by-side">
4     <dv.tx>
5       <h2>login</h2>
6       <authentication.sign-in /><dv.link href="/top" hidden=true />
7     </dv.tx>
8     <dv.tx>
9       <h2>register</h2>
10      <authentication.register-user /><dv.link href="/top" hidden=true />
11    </dv.tx>
12  </main>
13 </dv.component>

```

The screenshot displays the landing page of TopMovie. At the top is a red header with the text "topmovie" in white. Below the header, the page is split into two columns. The left column is titled "login" and contains two input fields: "Username \*" with the value "ben" and "Password \*". Below these fields is a "Sign In" button. The right column is titled "register" and contains three input fields: "Username \*" with the value "ben", "Password \*" with masked characters ".....", and "Retype Password \*" with masked characters ".....". Below these fields is a "Register User" button.

Figure 4-7: The code for the landing page of *TopMovie*

```

1 <dv.component name="top-movies">
2   <nav><h1>topmovie</h1></nav>
3   <main class="lr-margin">
4     <authentication.logged-in />
5     <dv.tx>
6       <dv.gen-id />
7       <movie.create-object id=dv.gen-id.id buttonLabel="Submit Movie" />
8       <scoring.create-score value=0 hidden=true
9         sourceId=authentication.logged-in.user?.id targetId=dv.gen-id.id />
10      <authentication.authenticate id=authentication.logged-in.user?.id
11        hidden=true />
12      <dv.link />
13    </dv.tx>
14    <scoring.show-targets-by-score showAscending=false
15      noTargetsText="No movies yet"
16      showTarget=<topmovie.show-movie target=$target
17        user=authentication.logged-in.user /> />
18  </main>
19 </dv.component>

```

# topmovie

Title \*

King's Speech

---

Submit Movie

- ▲ Titanic | 3 points | submitted 2 minutes ago
- ▲ Gladiator | 2 points | submitted 3 minutes ago
- ▲ Interstellar | 0 points | submitted a few seconds ago

Figure 4-8: The code for the top movies page of *TopMovie*



```

1 <dv.component name="show-movie">
2   <div class="movie">
3     <dv.tx>
4       <scoring.create-score submitMatIconName="arrow_drop_up"
5         sourceId=$user.id targetId=$target.id
6         showDoneMessage=false showOptionToInputValue=false value=1 />
7       <authentication.authenticate id=$user.id hidden=true />
8       <dv.link />
9     </dv.tx>
10    <div class="movie-info">
11      <movie.show-object id=$target.id includeTimestamp=true
12        hidden=true />
13      <strong>{{movie.show-object.loadedObject?.title}}</strong>
14      | <em>{{ $target.total }} points</em> | submitted <dv.show-date
15        date=movie.show-object.loadedObject?.timestamp format="time-ago" />
16    </div>
17  </div>

```

Figure 4-9: The code for the show movie component of *TopMovie*

loads the movie object (lines 11-12) and shows the loaded movie title (line 13). It also shows the movie points (line 14) and lets the logged-in user upvote the movie (lines 3-9). To restrict upvoting to authenticated users only, we synchronize `create-score` (line 4-6) with `authenticate` (line 7).

## 4.2.2 Discussion

The deep integration of scoring, posting, and authentication makes it hard to build *TopMovie* using a content management system. We can include an upvote plug-in, but we want the same users that upvote posts to be the ones that create posts, and we want the post creator to not be able to upvote their own posts. In a content management system, implementing this deep integration would typically require writing server-side code. Since general-purpose tools and end-user development tools are more flexible, deeply integrating scoring, posting, and authentication functionality is not a problem. The problem is that we would need to implement most of this functionality from scratch. Most end-user tools have some built-in authentication mechanism so, at the very least, authentication functionality wouldn't have to be implemented from scratch

in an end-user development tool, but upvoting and posting movie titles typically would.

## 4.3 Implementation of Custom Behavior

### 4.3.1 Example Application: FamilyLog

In *FamilyLog*, kids can share what they are doing with their parents and siblings. There are two kinds of users in *FamilyLog*: parents and children. Children can't create accounts directly, only parents can create accounts for their children. After a parent creates a child account, the new child account is subsequently associated with the parent account, and the child is added to the parent's family. A parent can see all log entries posted by their children. A child can log in to the application, submit new log entries, and see all log entries submitted by them or by their siblings.

The entire code for the Déjà Vu implementation of *FamilyLog* is shown in Figs. 4-10 through 4-14. The application has three web pages: a landing page (Fig. 4-12), a parent home page (Fig. 4-13), and a child home page (Fig. 4-14). The configuration file (Fig. 4-10) specifies the concepts we use in our application and the routes. In *FamilyLog* we use *Property* to save log entries, *Group* to associate parents with their children, and *Authentication* twice: once for parent accounts, and another for children accounts.

At first, it might appear that implementing the functionality that parents create accounts for their children would require a very advanced authentication concept, but Déjà Vu's composition mechanism by action synchronization and identifier sharing allows custom behavior like this to be implemented by combining multiple instances of the same simple authentication concept. In *FamilyLog*, we include two instances of *Authentication*, one for dealing with parent accounts (line 18) and the other one for children accounts (line 19). Parent accounts are registered with the `register-user` component of the parent authentication instance (Fig. 4-12, line 12). In the parent home page (Fig. 4-13), to let a parent create a child accounts, we synchronize the

```

1 {
2   "name": "familylog",
3   "usedConcepts": {
4     "entry": {
5       "name": "property",
6       "config": {
7         "schema": {
8           "title": "Entry", "type": "object",
9           "properties": {
10            "content": { "type": "string" },
11            "author": { "type": "string" },
12            "parentId": { "type": "string" }
13          },
14          "required": ["content", "author", "parentId"]
15        }
16      }
17    },
18    "parentauth": { "name": "authentication" },
19    "childauth": { "name": "authentication" },
20    "group": {}
21  },
22  "routes": [
23    { "path": "/", "component": "landing" },
24    { "path": "/parent", "component": "parent-home" },
25    { "path": "/child", "component": "child-home" }
26  ]
27 }

```

Figure 4-10: The configuration file of *FamilyLog*

```

1 @import "~@deja-vu/themes/scss/basil-green.scss";
2
3 .side-by-side {
4   display: flex;
5   justify-content: space-evenly;
6 }
7
8 nav h1 {
9   text-align: center;
10  font-size: 40px;
11 }
12
13 .log { width: 60%; }

```

Figure 4-11: The style sheet file of *FamilyLog*

`register-user` component of the child authentication instance (line 17) with the `authenticate` component of the parent instance (line 15). As a result, only authenticated parents can create child accounts.

To associate a parent account with its children we include *Group* (Fig. 4-10, line 20). Then, we wrap the `register-user` component for parents in a transaction with `create-group` (Fig. 4-12, line 11). As a result, when a new parent account is created, a group with the same ID as the parent account is created as well. Then, when a new child account is created we add the new child account to the parent group using `add-to-group` (Fig. 4-13, line 19).

To implement the log entry functionality we use *Property*. In this case, we configure property to save log entries: objects with a content and an author. We also include a `parentId` field to make it easy to fetch all the entries of a parent's children in the parent home page (Fig. 4-13, line 9) and in the child home page (Fig. 4-14, line 12). In the child home page, to retrieve the parent associated with the logged-in child, we find all groups that have the current logged-in child as a member (Fig. 4-14, line 5-6). Because of the way *Group* is used in *FamilyLog*, we know that a child can only belong to one group, which is the family group the child is part of. Thus, `show-groups` will return exactly one group ID in the `groupIds` output, which we access with the expression `group.show-groups.groupIds[0]`. We use the group ID

```

1 <dv.component name="landing">
2   <nav><h1>Family Log</h1></nav>
3   <main class="side-by-side">
4     <dv.tx>
5       <h1>Parent Login</h1>
6       <parentauth.sign-in /><dv.link href="/parent" />
7     </dv.tx>
8     <dv.tx>
9       <h1>New Parent</h1>
10      <dv.gen-id />
11      <group.create-group id=dv.gen-id.id hidden=true />
12      <parentauth.register-user id=dv.gen-id.id />
13      <dv.link href="/parent" />
14    </dv.tx>
15    <dv.tx>
16      <h1>Child Login</h1>
17      <childauth.sign-in /><dv.link href="/child" />
18    </dv.tx>
19  </main>
20 </dv.component>

```

The screenshot displays the 'Family Log' landing page with a light green background. It features three distinct login sections arranged horizontally:

- Parent Login:** Includes a 'Username \*' field with the value 'ben' and a 'Password \*' field. A 'Sign In' button is located below the password field.
- New Parent:** Includes a 'Username \*' field with the value 'ben', a 'Password \*' field, and a 'Retype Password \*' field. A 'Register User' button is highlighted with a blue border at the bottom of this section.
- Child Login:** Includes a 'Username \*' field and a 'Password \*' field. A 'Sign In' button is located below the password field.

Figure 4-12: The code for the landing page of *FamilyLog*

```

1 <dv.component name="parent-home">
2   <nav><h1>Family Log</h1></nav>
3   <parentauth.logged-in />
4   <main class="side-by-side">
5     <div class="log">
6       <h2>Log</h2>
7       <entry.show-objects showExclude=['parentId'] includeTimestamp=true
8         fieldMatching={
9           parentId: parentauth.logged-in.user?.id, waitOn: ['parentId']
10        } />
11     </div>
12     <dv.tx>
13       <h2>New Child</h2>
14       <dv.gen-id />
15       <parentauth.authenticate user=parentauth.logged-in.user
16         hidden=true />
17       <childauth.register-user id=dv.gen-id.id
18         signIn=false buttonLabel="Create Child" />
19       <group.add-to-group id=parentauth.logged-in.user?.id
20         memberId=dv.gen-id.id hidden=true />
21     </dv.tx>
22 </dv.component>

```

Figure 4-13: The code for the parent home page of *FamilyLog*

```

1 <dv.component name="child-home">
2   <nav><h1>Family Log</h1></nav>
3   <main>
4     <childauth.logged-in />
5     <group.show-groups hidden=true waitOn=['withMemberId']
6       withMemberId=childauth.logged-in.user?.id />
7     <dv.if condition=group.show-groups.groupIds class="side-by-side">
8       <div class="log">
9         <h2>Log</h2>
10        <entry.show-objects showExclude=['parentId'] includeTimestamp=true
11          fieldMatching={
12            parentId: group.show-groups.groupIds[0], waitOn: ['parentId']
13          } />
14      </div>
15      <dv.tx>
16        <h2>Create Entry</h2>
17        <entry.create-object showExclude=['author', 'parentId']
18          initialValue={
19            author: childauth.logged-in.user?.username,
20            parentId: group.show-groups.groupIds[0] } />
21        <childauth.authenticate user=childauth.logged-in.user
22          hidden=true />
23        <dv.link />
24      </dv.tx>
25    </dv.if>
26  </main>
27 </dv.component>

```

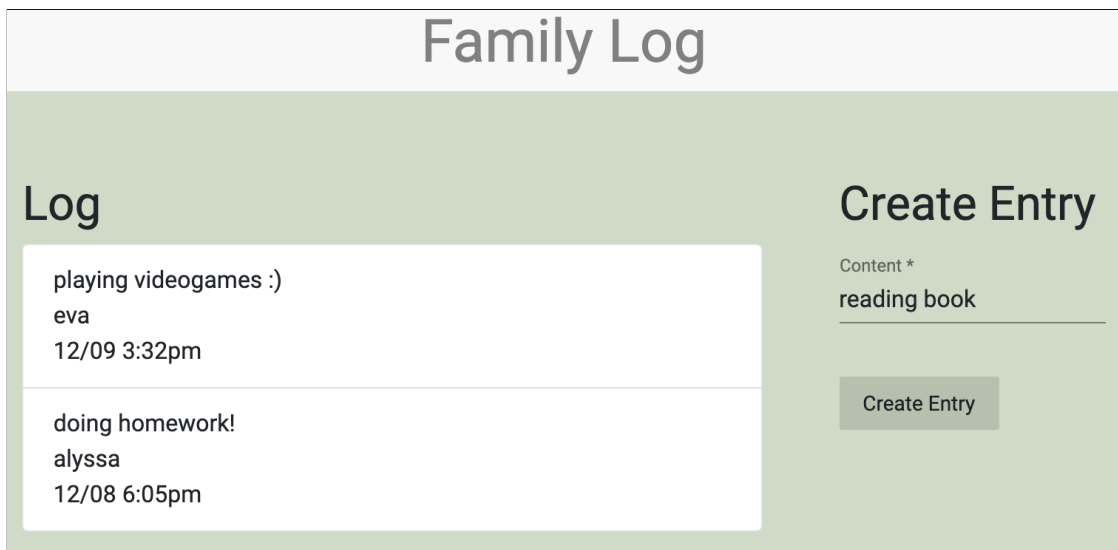


Figure 4-14: The code for the child home page of *FamilyLog*

as the parent ID to fetch the log entries (line 12) and to add a parent ID to new log entries (line 20).

### 4.3.2 Discussion

The *FamilyLog* application implements custom behavior in which we have one type of user that can create accounts for another type of user. In content management systems and end-user programming tools, implementing this behavior requires advanced built-in authentication functionality or the existence of a plug-in that supports having two user types in which one user type creates accounts of the other. In *Déjà Vu* however, it is possible to implement the functionality by including a simple authentication concept twice and by wrapping certain actions in a transaction. This example shows how flexible our composition mechanism can be. With our composition mechanism, custom behavior can be implemented without requiring the implementation of a concept that is very specific to the application.

## 4.4 Benefits of Concept Modularity

To illustrate the benefits of concepts compared to conventional libraries and software architectures, let's consider a small example component that lets users create posts with a rating and see what it looks like to implement the component using conventional general-purpose tools and using *Déjà Vu*.

Fig. 4-15 shows `submit-post` written in Angular and React. For the example, we assume that we have a `post-input` component that lets the user input the content of the post, and a `rating-input` component that lets the user select a star rating for the post (Fig. 4-15a, template file, lines 2-3 and Fig. 4-15b, lines 18-19). We also assume that we have a client-side post service library for making requests (Fig. 4-15a, component file, line 6 and Fig. 4-15b, line 12).

Fig. 4-16 shows the server-side code of `submit-post`. We consider two variants for the server-side code: as a monolith and using microservices. We assume that the server will process client requests and invoke the `savePost` server-side function



```

1 <form (ngSubmit)="submitPost()">
2   <app-post-input [ngModel]="p"></app-post-input>
3   <app-rating-input [ngModel]="r"></app-rating-input>
4   <button type="submit">Submit</button>
5 </form>

```

```

1 @Component({ selector: 'app-submit-post', ... })
2 export class SubmitPostComponent {
3   p: Post; r: number;
4   constructor(private postService: PostService) {}
5   submitPost() {
6     this.postService.savePost(this.p, this.r);
7   }
8 }

```

(a) submit-post component in Angular

```

1 class SubmitPost extends React.Component {
2   constructor(props) {
3     super(props);
4     this.state = { p: new Post(), r: 0 };
5     this.handleChange = this.handleChange.bind(this);
6     this.handleSubmit = this.handleSubmit.bind(this);
7   }
8   handleChange(e) {
9     this.setState({ [e.target.name]: e.target.value });
10  }
11  handleSubmit(e) {
12    PostService.savePost(this.state.p, this.state.r);
13    e.preventDefault();
14  }
15  render() {
16    return (
17      <form onSubmit={this.handleSubmit}>
18        <PostInput onChange={this.handleChange} />
19        <RatingInput onChange={this.handleChange} />
20        <button type="submit">Submit</button>
21      </form>
22    );
23  }
24 }

```

(b) submit-post component in React

Figure 4-15: Two client-side code variants for submit-post: using Angular and React

```

1 function savePost(p: Post, r: number) {
2   if (Post.isValid(p) && Rating.isValid(r)) {
3     db.save(p);
4   }
5 }

```

(a) Server Monolith

```

1 function savePost(p: Post, r: number) {
2   if (PostService.isValid(p) &&
3     RatingService.isValid(r)) {
4     const newP = PostService.newPost(p);
5     RatingService.newRating(newP.id, r);
6   }
7 }

```

(b) Server using Microservices

Figure 4-16: Two server-side code variants for `submit-post`: using a monolithic architecture and microservices

```

1 <dv.component name="submit-post">
2   <dv.tx>
3     <dv.gen-id />
4     <property.create-object id=dv.gen-id.id />
5     <rating.rate-target targetId=dv.gen-id.id />
6     <dv.button>Submit</dv.button>
7   </dv.tx>
8 </dv.component>

```

Figure 4-17: `submit-post` component in *Déjà Vu*

with the correct inputs (Fig. 4-16a or Fig. 4-16b, line 1). In the monolithic back-end (Fig. 4-16a), we assume there are libraries for validating posts, ratings, and accessing the database. In the microservices back-end (Fig. 4-16b), we assume there are post and rating services.

Note that, even if we assume that all the post and rating functionality is readily available, there's still a lot of code that we need to write to put everything together. With a standard approach, we have to:

- *Subscribe to events and write event handlers.* In Angular, we subscribe to the `ngSubmit` event generated by the form so as to invoke the `onSubmit` method of the component whenever the form gets submitted (Fig. 4-15a, template file, line 1). In React, we subscribe to the `onChange` event generated by the post and rating components (Fig. 4-15b, lines 18-19) and to the `onSubmit` event of the form (Fig. 4-15b, line 17).
- *Aggregate data for the request.* In Angular, the event handler for submit events, `submitPost()`, has to aggregate data of `post-input` and `rating-input` to build the server request. In React, the event handler for `onChange` events updates the component state, which is then read by the event handler for submit events to build the server request. In both cases, the request is sent by the post service when we invoke its `savePost` method.
- *Handle client-server communication.* While we are assuming that there exists a client-side post service to issue requests, certain modifications would normally require the modification of the server API and of the client service. For example, adding a location to the post might require adding a new parameter to `savePost` and a new parameter to the server API so that it expects a location value.
- *Combine server-side functionality.* Even if the server-side functionality of posting and rating exists, we still have to, in the case of the monolith, make the appropriate function calls to validate the request and save the post to the database. Also, in the server using microservices, we still have to make the appropriate

calls to the post and rating services. Moreover, integrating microservices could, in some cases, require more complex code, since one might have to implement transactions.

On the other hand, when using Déjà Vu, none of this is necessary. Fig. 4-17 shows the implementation of `submit-post` using Déjà Vu, where we assume that we have the *Property* and *Rating* concepts. In Déjà Vu, wrapping a component in a `dv.tx` automatically subscribes to run events, runs the event handlers, aggregates the data client-side and sends the request, unpacks the request server-side, and coordinates the calls to the back-end services of each concept so they happen in a transaction if necessary. All of this functionality is hidden from the developer, who doesn't need to write JavaScript code to handle events or combine server-side microservices for example. Of course one could build a set of libraries that would alleviate the amount of integration code required when using a standard approach, but that would amount to almost replicating Déjà Vu's runtime system and our concept modularity.

## 4.5 Summary

Through three small example applications, *SecretParty*, *TopMovie*, and *FamilyLog*, we have shown how it is possible to use Déjà Vu to build applications with rich graphical user interfaces and complex behavior. Since concepts implement full-stack end-user behavior, it is possible to quickly add new functionality to an application by including a concept. Including a concept only requires writing a little JSON and some HTML to include concept components. Déjà Vu's composition mechanism allows the developer to deeply integrate different end-user functionality, by sharing identifiers and synchronizing actions in HTML, without writing any server-side code or JavaScript code. The composition mechanism is flexible enough that implementing behavior that might, at first, seem like it would require a very application-specific concept, can often be implemented by combining existing concepts.

Also, we have shown how our concept modularity provides benefits compared to conventional libraries and software architectures by abstracting away complicated

client-server integration code, such as subscribing to client-side events, preparing requests, and combining server-side functionality.

In the next chapter, we are going to explain in detail how to build applications using Déjà Vu with a longer example application.

THIS PAGE INTENTIONALLY LEFT BLANK

# Chapter 5

## Building Applications with Déjà Vu

To see what using Déjà Vu is like, let's consider building an application called *Slacker News (SN)*. *Slacker News* is a simple clone of Hacker News<sup>1</sup>, a popular social news aggregation website. In *Slacker News*, registered users can post links, comment on posts, and upvote posts or comments. Posts with the most upvotes appear towards the top of the home page. Figs. 5-1 through 5-4 show screenshots of what the application *Slacker News* looks like.

To build an application with Déjà Vu, you include and configure concepts (§5.1) and create application components by linking components (§5.2). To implement a reactive user interface, you use the same synchronization mechanism used for composing concept behavior (§5.3). The security policy of the application is specified implicitly in the component code and application configuration (§5.4). To customize the appearance of the application, you write CSS or SASS code (§5.5). Finally, while it not necessary to modify a concept implementation to build *Slacker News*, we conclude the chapter with an overview on how you can create a new concept or customize a concept implementation in Déjà Vu if you have to (§5.6).

**Slacker News** [Top](#) [New](#) | [Submit](#) [Login](#)

---

### Sign In

Username \*  
\_\_\_\_\_

Password \*  
\_\_\_\_\_

[Sign In](#)

### Register

Username \*  
**ben**  
\_\_\_\_\_

Password \*  
.....  
\_\_\_\_\_

Retype Password \*  
.....  
\_\_\_\_\_

[Register User](#)

Figure 5-1: Screenshot of the register user page of *Slacker News*

**Slacker News** [Top](#) [New](#) | [Submit](#) [ben](#) [Sign Out](#)

---

### Submit

Title \*  
Déjà Vu Platform  
\_\_\_\_\_

Uri \*  
<https://deja-vu-platform.com>  
\_\_\_\_\_

[Submit](#)

Figure 5-2: Screenshot of the submit post page of *Slacker News*



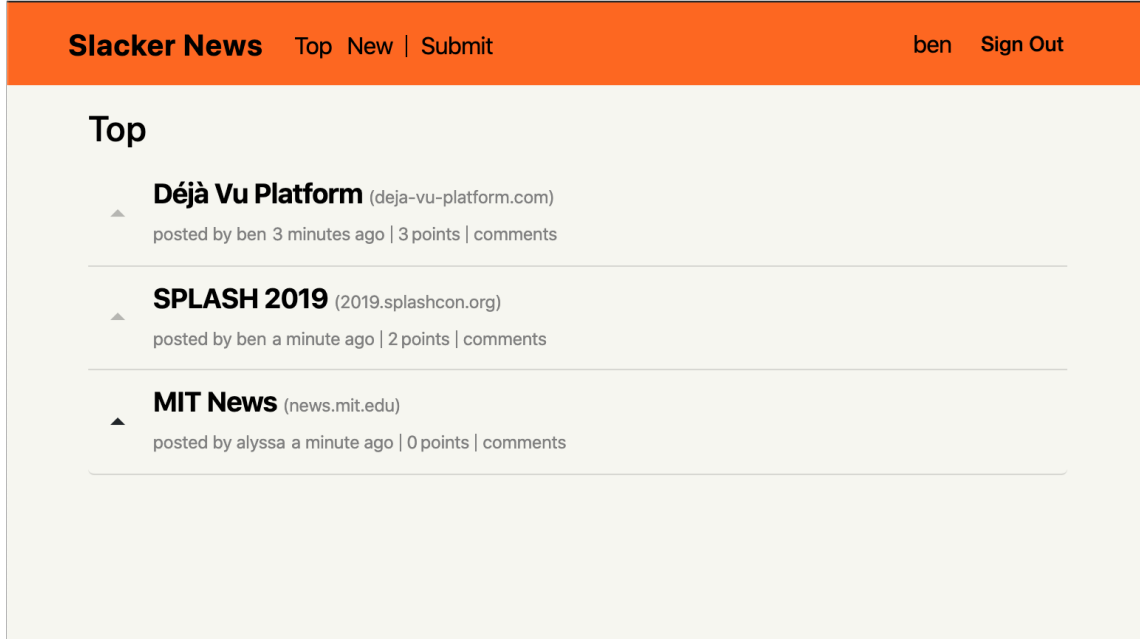


Figure 5-3: Screenshot of the home page of *Slacker News*

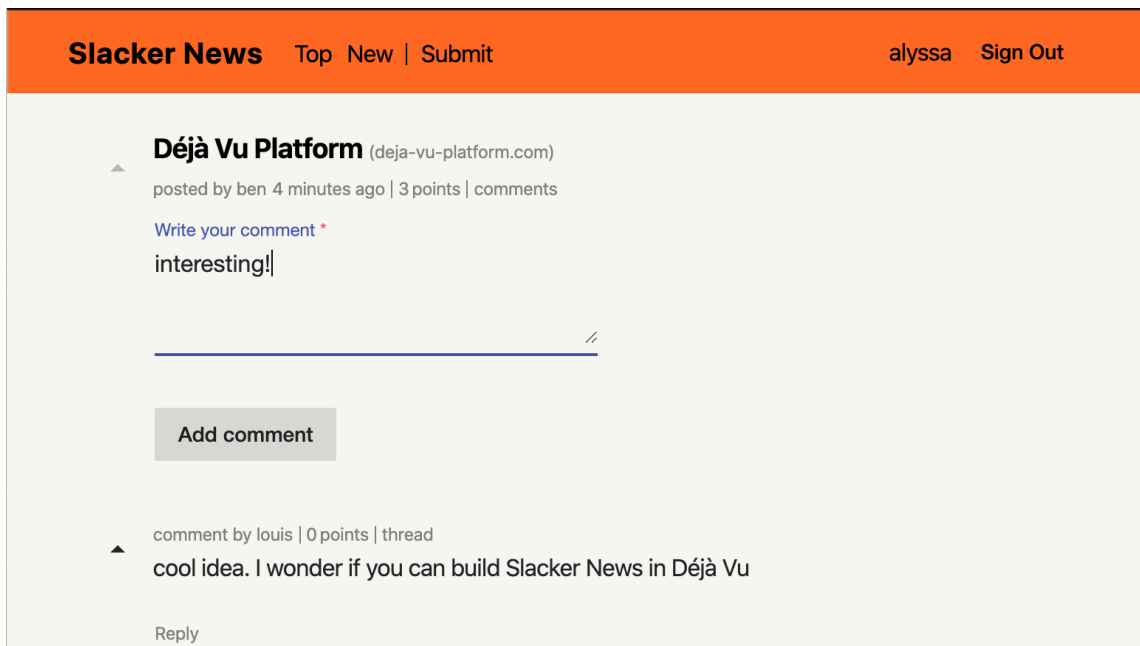
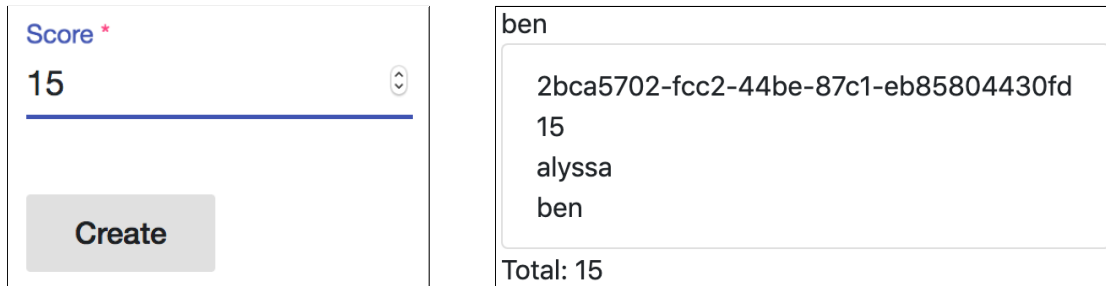


Figure 5-4: Screenshot of the post detail page of *Slacker News*



(a) `create-score` with `targetId` input “ben” and `sourceId` “alyssa” (b) `show-target` with `id` input “ben” showing the score created in Fig. 5-5a and ben’s total score

Figure 5-5: Screenshots of two components of *Scoring*

## 5.1 Including and Configuring Concepts

### 5.1.1 Choosing Concepts

The process of building a Déjà Vu application begins by navigating the catalog of concepts to find the concepts that provide the functionality you need for your application. The concepts in our catalog and their purposes are shown in Table 5.1. The documentation accompanying a concept includes information about the configuration options and the exported components. Fig. 5-5 shows some components of *Scoring*. Concept components control a patch of the screen, are interactive, and can read and write back-end data. They also have input and output properties.

*Slacker News* uses *Authentication* to handle user authentication, *Comment* to comment on posts and reply to comments, and *Scoring* twice: for keeping track of upvotes on both posts and on comments separately. It also uses *Property*, which, as we’ve seen in Chapter 4, provides a data-model-defining facility for simple CRUD behavior. In *Slacker News*, we use *Property* to save a post’s author, title, and URL.

### 5.1.2 Including Concepts

The concepts used by the application—*Authentication*, *Comment*, *Property*, and *Scoring* twice—are specified in the application’s JSON configuration file (Fig. 5-6, lines 3-22). The `usedConcepts` object has one key-value pair per concept instance. The

<sup>1</sup><https://news.ycombinator.com/>

```

1 {
2   "name": "sn",
3   "usedConcepts": {
4     "authentication": {},
5     "comment": {},
6     "post": {
7       "name": "Property",
8       "config": {
9         "schema": {
10          "title": "Post", "type": "object",
11          "properties": {
12            "author": { "type": "string" },
13            "title": { "type": "string" },
14            "url": { "type": "string", "format": "url" }
15          },
16          "required": [ "author", "title", "url" ]
17        }
18      }
19    },
20    "scoreposts": { "name": "Scoring" },
21    "scorecomments": { "name": "Scoring" }
22  },
23  "routes": [
24    { "path": "", "component": "home" },
25    { "path": "/login", "component": "login" },
26    { "path": "/post", "component": "show-post-details" },
27    { "path": "/comment", "component": "show-comment-details" },
28    { "path": "/new", "component": "new" }
29  ]
30 }

```

Figure 5-6: The configuration file of *Slacker News*

Table 5.1: Concept catalog

Concept	Purpose
Authentication	Verify a user’s identity with a username and password
Authorization	Control access to resources
Chat	Exchange messages in real time
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 they both agree
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 a time to meet
Scoring	Keep track of scores
Task	Keep track of pieces of work to be done
Transfer	Transfer money or items between accounts

key (e.g., “post” on line 6) determines the name that is going to be used in the HTML to refer to that instance. The value is another object with two optional key-value pairs: **name** for providing the name of the concept to be instantiated (e.g., “Property” on line 7), and **config** for specifying the configuring options for the concept instance (e.g., the object in lines 8-18). If no concept name is provided, the concept instantiated is the one with name equal to the instance name. Thus, for example, the concept to be instantiated for “authentication” is *Authentication* (line 4). If no configuration object is given, the default configuration for that concept is used. The format of the values of configuration options is also JSON.

### 5.1.3 Configuring Concepts

In *Slacker News*, we only have to configure *Property*. *Property* accepts a configuration variable **schema** that expects a JSON Schema<sup>2</sup> to describe the objects it will be saving.

<sup>2</sup><https://json-schema.org/>

We use `schema` to specify the type of properties we expect our objects to have. In *Slacker News*, our objects are “posts” (line 10), and we expect them to have an author, title, and a URL (lines 11-15). The effect of configuring *Property* like we do in the configuration file, is that when we include a component from *Property*, such as `create-object`, the component will allow the user to input only those fields—author, title, and URL. Moreover, since we specified that the format of the URL field is `url` (line 14) and that the fields author, title, and URL are required (line 16), `create-object` will expect the end-user to provide a value for each field and check that the value given for the URL field is a valid URL. If the end-user doesn’t provide a value for each field, or if the URL value given is invalid, `create-object` will show an error message.

#### 5.1.4 Other Application Configuration

In the configuration file of the application, we also define the name (Fig. 5-6, line 2) and routes (lines 23-29) of our application. Each route maps a URL path to a component. A component that is accessible via URL is a page. *Slacker News*’s home page is the component `home` (line 24) because `path` is empty. If the user navigates to `/login`, the `login` component will be shown (line 25), if the user navigates to `/post`, the `show-post-details` component will be shown (line 26), and so on.

## 5.2 Linking Components

Each application component is written in a separate HTML file. Fig. 5-7 shows an excerpt of the code for *Slacker News*’s `submit-post` component and Fig. 5-8 shows an excerpt of the code for *Slacker News*’s `show-post` component. The full code for the application is available online<sup>3</sup>. We also include, in each figure, a screenshot of how the component appears to end-users of the application.

---

<sup>3</sup><https://github.com/spderosso/deja-vu/tree/master/samples/sn>

```

1 <dv.component name="submit-post">
2   <sn.navbar /> ...
3   <dv.tx>
4     <dv.gen-id />
5     <authentication.authenticate
6       username=sn.navbar.user.username hidden=true />
7     <post.create-object id=dv.gen-id.id
8       initialValue={ author: sn.navbar.user.username }
9       showExclude=["author"] buttonLabel="Submit"
10      newObjectSavedText="Post submitted" />
11     <scoreposts.create-score targetId=dv.gen-id.id
12       sourceId=sn.navbar.user.username value=0 hidden=true />
13     <dv.link href="/post" params={ id: dv.gen-id.id } hidden=true />
14   </dv.tx> ...
15 </dv.component>

```

The screenshot shows a web interface for submitting a post. At the top, an orange navigation bar contains the text "Slacker News", "Top New | Submit", "ben", and "Sign Out". Below this is a white form area with the title "Submit". The form contains two text input fields: "Title \*" and "Url \*". At the bottom of the form is a "Submit" button. The form area is labeled "post.create-object" in the bottom right corner.

Figure 5-7: Excerpt of *Slacker News*'s submit-post component

```

1 <dv.component name="show-post"> ...
2   <post.show-object id=$id hidden=true /> ...
3   <dv.if condition=post.show-object.loadedObject>
4     <sn.upvote id=$id user=$user />
5     <a href=post.show-object.loadedObject.url>
6       {{post.show-object.loadedObject.title}}
7     </a> ...
8     (<post.show-url showBaseUrlOnly=true
9       url=post.show-object.loadedObject.url />)
10    posted by {{post.show-object.loadedObject.author}}
11    <dv.show-date format='time-ago'
12      date=post.show-object.loadedObject.timestamp /> ...
13    <scoreposts.show-target
14      id=$id showId=false showScores=false totalLabel="" /> points
15    <dv.link href="/post" params={ id: $id }>comments</dv.link>
16  </dv.if>
17 </dv.component>

```



Figure 5-8: Excerpt of *Slacker News*'s show-post component

Our template language looks, by design, similar to other template languages. To create an application component, a developer includes components (§5.2.1) and synchronizes components to implement the desired functionality (§5.2.2).

## 5.2.1 Including Components

### Including Components

An application component can contain other components, which can be concept components or application components. A concept component is a component that is defined and exported by a concept. An application component, on the other hand, is defined by the application developer and it is part of the application being developed. Components are included as if they were HTML elements, with the tag given by the concept instance or application name, followed by the component name. Thus, `submit-post` (Fig. 5-7) includes one application component, `navbar` (line 2); three concept components, `authenticate` of *Authentication* (lines 5-6), `create-object` of the `post` instance of *Property* (lines 7-10), and `create-score` of the `scoreposts` instance of *Scoring* (lines 11-13); and two built-in components, `dv.gen-id` (line 4) and `dv.link` (line 13).

### Input/Output Binding

Inputs to a component are bound with the syntax `property=expr`. Template expressions can include literal values, application component inputs, outputs from included components, and standard operators. No cycles in the bindings are allowed. The syntax of template expressions is similar to that of JavaScript expressions, but no function calls or JavaScript operators that produce side-effects are allowed.

Components can be fired repeatedly, and the output properties hold the values from the last execution. This is how a selector widget such as a dropdown would typically be connected to another component: the dropdown sets an output property every time it is activated containing the choice the user made, which is then bound to the input property of components that use that choice.



Some input properties are for customizing appearance and have no impact on the behavior of the component. For example, as a result of setting `buttonLabel` to "Submit" (Fig. 5-7, line 9), `create-object`'s button will carry the label "Submit" instead of the default button label "Create Post". The hidden property of `show-object` (Fig. 5-8, line 2) indicates that the component should be activated but not visible. Thus the object data itself is still loaded, emitted as an output, and used in several parts of the view—the title and the URL are used and shown through lines 5-9, the author is shown on line 10, and the object creation timestamp is shown on line 11-12.

Application components can have their own input properties. Any name used in an expression that is prefixed with `$` is considered to be an input property of the application component. For example, `show-post` (Fig. 5-8) has an input named `id` that it uses in lines 2, 4, 14, and 15. Based on this input, `show-object` will show the post whose ID matches the given one; `upvote` will use the ID as the target of the score if one is created; `show-target` will show the score with the given ID; and clicking on the "comments" link will take the user to `show-post-details` with its input `id` set to the given ID.

To display how long ago the post was created in `show-post`, we bind the `date` input property of `show-date` to the object creation timestamp (line 12) and set its `format` input property to "time-ago" (line 11). All objects from *Property* have a `timestamp` field that stores the object's creation date.

## Identifier Sharing

To bind entities in different concepts we use a common identifier. In `submit-post` (Fig. 5-7), for example, the same ID, generated by `gen-id` (line 4), is passed to `create-object` (line 7), `create-score` (line 11), and `dv.link` (line 13). As a result, `create-score` will create a score with the same target ID as the object created by `create-object`, and `dv.link` will cause a redirect to the "/post" page with its `id` input set to the new post ID after the post submission succeeds. Similarly, in `show-post` (Fig. 5-8), we feed the `id` input to `show-object` (line 2) and `show-target`

(line 14). Each of these components loads and displays its own view of the post entity; the effect when put together is to display a *Slacker News* post object.

## 5.2.2 Synchronizing Components

### Action Types

Concept components have at most two server-side actions: an evaluation action (eval) and an execution action (exec). The concept author determines what triggers the evaluation or execution of the component. Typically, the loading of the component into the user's web browser triggers the evaluation of a component, and some user interaction, such as a button click, triggers its execution. What happens on eval or exec is also up to the author of the concept—the only restriction is that an eval action cannot produce a side effect on the server state. Note that application components don't have actions. This is because application components have no back-end functionality of their own—all data and behavior is pushed to concepts.

Eval/exec actions support the conventional user interaction pattern of web applications: data is loaded and displayed, and then the user executes commands to mutate the data. It would be possible for concept components to offer arbitrary action types to support more complex forms of behavior. But this would require more work from the user, who would now have to specify what action types are to be coordinated.

### Synchronizing Actions

There are two kinds of application components: a regular component and a transaction (tx) component. A regular component allows any of its children components to eval/exec without synchronization. A transaction component, on the other hand, synchronizes the actions of the concept components it wraps, so that an exec in one happens with the exec of the other(s)—and similarly for eval actions. Synchronized actions either complete in their entirety if all succeed, or have no effect whatsoever other than optionally displaying an error if one or more action aborts. Instead of putting each component in separate HTML files, you can wrap elements in another

```

1 <dv.component name="upvote"> ...
2   <dv.tx>
3     <authentication.authenticate
4       username=$user.username hidden=true />
5     <scoringposts.create-score
6       value=1 sourceId=$user.username targetId=$id ... />
7     <dv.link hidden=true />
8   </dv.tx> ...
9 </dv.component>

```

Figure 5-9: Excerpt of *Slacker News*'s upvote component

component with the `dv.tx` tag to create an anonymous transaction component with content equal to the content of the tag.

In *Slacker News*'s `submit-post`, the transaction is triggered by `create-object` (Fig. 5-7, line 7) when the user clicks on the “Submit” button. This is because the button in the `create-object` component of *Property* causes the component to execute on click, and since `create-object` is wrapped in a `dv.tx`, it will trigger the execution of all its sibling concept components. As a result, a new post and a new score will be created, bound by the shared ID. The inclusion of `authenticate` in the transaction prevents an unauthenticated user from creating a post. The inclusion of `dv.link` in the transaction causes a redirect after the transaction succeeds.

## 5.3 Building Reactive User Interfaces

Typical modern web applications react immediately to user interactions, without requiring the user to refresh the web page. For example, on the Twitter home page there's a form to submit a tweet above your feed. If you submit a tweet, the feed refreshes automatically to show your new tweet.

To build reactive user interfaces such as the Twitter interface in *Déjà Vu*, a developer uses the `dv.link` component and transactions. For example, in the home page of *Slacker News* (Fig. 5-3), we want to automatically refresh the list of posts after the end-user upvotes a post. Fig. 5-9 shows how we implement this reactive behavior

in the `upvote` component of *Slacker News*. In the `upvote` transaction, we include a hidden `dv.link` (line 7) with no `href` value. The effect of doing so is that `dv.link` will cause a redirect after the transaction succeeds and `create-score` records the new vote. Since no `href` value is provided to `dv.link`, the `dv.link` component refreshes the current page. *Déjà Vu* applications are single-page applications that dynamically rewrite the current page, rather than loading entire new pages from the server. Therefore, the refresh caused by `dv.link` doesn't trigger a full-reload of the page, but instead prompts components in the page to re-fetch data from the server and update the graphical user interface to reflect the new data.

Some modern user interfaces not only refresh after a user interaction but can also refresh after a server-side event happens, such as a data update produced by another user. The canonical example of this behavior is chat functionality. In a chat room, as soon as a new message is posted to the chat room by another user, the current page refreshes to show the new message.

In *Déjà Vu*, the components of the *Chat* concept automatically refresh themselves when server data changes. The developer doesn't have to do anything other than include a chat component as it would include any other component. As of this writing, only *Chat* includes components that automatically refresh when server data changes, but we expect all concept components to eventually support automatic refreshing. Our platform already has the necessary infrastructure for concept authors to implement components that can react to server-side events.

## 5.4 Specifying Security Policies

In *Déjà Vu*, a security policy is specified implicitly through: (1) which concepts are included and how they are configured; (2) which components from the included concepts are used; and (3) how concept components are bound to other concept components in transaction components.

For example, to implement the policy that posts must have a title, we say that the title field is required in the configuration of *Property* (Fig. 5-6, line 16). The server

of *Property* will enforce this constraint, and return an error if no title is given when a new object is created. Other constraints, such as the constraint that posts cannot be deleted, are enforced by the omission of certain components. In this case, the `delete-object` component that lets end-users delete *Property* objects is not included in the application. Finally, Fig. 5-9 shows how we use transaction components to implement the policy that only authenticated users can upvote posts. In *Slacker News*'s `upvote` component, we wrap the creation of a new score in a transaction component with the `authenticate` component of *Authentication*. The `authenticate` component checks that the logged-in user matches the given username. If it doesn't, it returns an error. The error causes the transaction to abort and, as a result, it causes the upvote of a post to abort. Since there's no other component in the application that would let a user upvote a post, only authenticated users can upvote a post. We also use `authenticate` in `submit-post` (Fig. 5-7, lines 5-6) to prevent unauthenticated users from creating new posts.

Note that policies are expressed in HTML and JSON, but, as we'll see later in Chapter 7, are actually enforced server-side by the platform's runtime system and concept servers.

## 5.5 Styling the Application

The appearance of a Déjà Vu application can be customized using CSS or SASS. A developer can define a global style sheet file. The rules in the global style sheet apply to all components of the application. A developer can also provide a style sheet for each individual component that applies only to that component.

While concept components come with some default styling, they also export a set of CSS classes for developers to overwrite the default style. A developer can refer to these CSS classes in the global style file to style HTML elements inside concept components. The CSS classes exported by concept components are named using the convention `conceptname-componentname-element`. For example, in *Slacker News*'s global style file, we write a rule to match the `username` element of the `show-user`

```

1 $navbarButtonPadding: 0 5px;
2
3 @import "~@deja-vu/themes/scss/orange-and-black.scss";
4
5 .authentication-show-user-username {
6   display: inline;
7 }
8 ...
9 .sep {
10  border-left: 1px solid gray;
11  width: 1px;
12  height: 12px;
13  margin-top: 3px;
14  margin-left: 4px;
15  margin-right: 4px;
16 }
17 ...

```

Figure 5-10: Excerpt of *Slacker News*'s global style sheet file

component of `authentication` and have it display inline (Fig. 5-10, lines 5-7). This rule applies globally to all `username` elements of any `show-user` component on the application. If we include `show-user` more than once in the application and we would like to style each component instance differently, we can use CSS to distinguish between the different component instances and style them differently. For example, let's say we have an application with two instances of `show-user` that we want to style differently, one in the navbar and another in the profile page. We can add a CSS class `show-user-nav` to the `show-user` element in the navbar and a CSS class `show-user-profile` to the `show-user` element in the profile page. Then, in the global style file, we create two rules: one that matches the `username` element that is a descendant of the `show-user` element with class `show-user-nav` and the other rule that matches the `username` element that is a descendant of the `show-user` element with class `show-user-profile`.

A developer can, of course, also style HTML elements that appear in application components. For example, in *Slacker News*'s global style sheet (Fig. 5-10) we match HTML elements with the class `sep` to give it a left border, a certain width, height

and margin (lines 9-16). We use `sep` in *Slacker News* to create the vertical bars that separate post information such as the date and author in, for example, `show-post` (Fig. 5-8).

### 5.5.1 Themes

To make it easier for developers to quickly style an application, our platform includes a set of themes. The themes style HTML elements such as `<nav>`, `<main>`, and `<p>`. Themes export set of variables that the developer can override. For example, in *Slacker News* we use the orange and black theme (Fig. 5-10, line 3), but we override the default value for the `$navbarButtonPadding` to be `0 5px` (line 1).

## 5.6 Customizing a Concept Implementation

For building *Slacker News* we assumed that all the functionality we required was already there in the catalog. But what is the procedure if, for example, there's a concept you need that is not in the catalog, or if you need a new component that is not present in a concept?

To implement or modify a concept you use general-purpose tools. In particular, you use the Angular front-end web framework, and the TypeScript language for both client- and server-side programming. All our concepts are open-source and the concept code can be modified and repackaged for use in an application.

Creating or modifying a concept is no different than creating or modifying the code for a regular web application built with modern web frameworks and tools. To create a concept component you create an Angular component, which consists of an HTML template, CSS, and an associated TypeScript class. In the TypeScript class, you import a *Déjà Vu* library to subscribe to `eval/exec` events and to run the server-side action of the component. To create a server-side action you modify a server file.

To make it easy to author concepts our platform includes several libraries for implementing transactions and processing client-side requests. We also have a command-

line interface tool for scaffolding concept implementations. At this point however, our concepts abstractions are completely opaque and we have no platform mechanism that allows a developer to easily override the behavior of a concept without having to read and modify concept code. In Chapter 10, we discuss potential improvements to our platform to make it easier for a developer to build and modify concepts.

## 5.7 Summary

To build applications with Déjà Vu a developer navigates the concept catalog to find the functionality they need, writes a JSON configuration file to configure and include concepts, and writes an HTML file for each application component. In each HTML file, the developer can include concept or application components, bind their inputs/outputs together, and specify whether the server-side actions of the included concept components should be synchronized or not. To style the application, the developer writes CSS or SASS, and can use one of our built-in themes.

In the next chapter, we give a formal semantics to Déjà Vu to better understand and communicate the behavior of Déjà Vu applications.



# Chapter 6

## Platform Semantics

In this chapter, we give a formal semantics to Déjà Vu. The purpose of giving a formal semantics to Déjà Vu is to build a mathematical model that can serve as a basis for understanding and reasoning about the behavior of a Déjà Vu application. We give a formal semantics to Déjà Vu to communicate, precisely, the behavior of a Déjà Vu application.

Two main approaches for specifying a formal semantics are operational and denotational semantics. An operational semantics describes the meaning of a language by specifying how it executes on an abstract machine. A denotational semantics, on the other hand, describes the meaning of a language by constructing mathematical objects that represent what the program does and translating the language to these objects.

The operational and denotational style are not in opposition to each other and combining them can be beneficial. In our formal semantics of Déjà Vu, we combine both approaches. We define a set of mathematical objects that correspond to relevant notions in Déjà Vu, such as components, concept server states, applications, and so on, and use operational semantics to describe the meaning of a Déjà Vu application as a set of computational steps that a Déjà Vu application can perform. Then, in a denotational style, we translate a core Déjà Vu syntax to the set of mathematical objects we defined previously. This separation allows us to use high-level mathemat-

ical objects to understand the computational behavior of a Déjà Vu application and abstract away compilation details.

## 6.1 Introduction

To give an operational semantics to Déjà Vu we use the framework of structural operational semantics [52]. The structural operational semantics of Déjà Vu defines transitions between configurations. A configuration represents the state of execution of an application instance at a given moment. The semantics of an instance of a Déjà Vu application is a set of finite but unbounded sequences of transitions between application instance configurations. We say the transition sequence is finite but unbounded because a deployed Déjà Vu application runs until it is terminated by the developer—a Déjà Vu application won't terminate by itself after a series of steps.

To define the set of possible transitions we define a transition relation. We specify the transition relation using inference rules that define the transition of an application instance configuration in terms of the transition of its parts: clients, components, and concept server states.

We are going to explain the semantics incrementally. On each iteration, we refine the definition and rules introduced in the previous iteration of the semantics and add new definitions and rules. The first iteration of the semantics is concerned only with components that can propagate input/output changes through the property bindings and read/write concept state (§6.2). The second iteration adds support for multiple clients (§6.3). In the third and last iteration, we add action synchronization to give the full semantics of Déjà Vu (§6.4). We then present a core syntax of Déjà Vu and its translation to the semantic elements used in the operational semantics (§6.5), and conclude the chapter with a discussion on minor features of Déjà Vu that we chose to omit from the formal semantics (§6.6).

## 6.2 First Iteration: Components

In the first iteration of the semantics, we are concerned only with components and the global server state of an application instance. We begin by defining property bindings, component configurations, and application instance configurations (§6.2.1). We then specify the behavior of concept components (§6.2.2), give the inference rule that determines the behavior of an application instance configuration (§6.2.3), and discuss how the initial application instance configuration of an application is determined (§6.2.4).

### 6.2.1 Definitions

We start by defining the following sets:

- $\mathcal{T}$ ,  $\mathcal{N}$ , and  $\mathcal{P}$  are the set of concept names, component names, and property names respectively. Input, output, and private properties are properties with names in  $\mathcal{P}_{in}$ ,  $\mathcal{P}_{out}$ , and  $\mathcal{P}_{private}$  respectively. Each property name is an input, output, or private property name. That is,  $\mathcal{P} = \mathcal{P}_{in} \dot{\cup} \mathcal{P}_{out} \dot{\cup} \mathcal{P}_{private}$ . We use  $A \dot{\cup} B$  to denote the union of disjoint sets  $A$  and  $B$  ( $A \cap B = \emptyset$ ).
- The set  $\mathcal{V}$  is the set of property values and includes numbers, booleans, strings, records, arrays, components, and  $\perp$ , which denotes the absence of value.
- $\mathcal{S}$  is the set of concept server states.
- $\mathcal{I} \subseteq (\mathcal{T} \times \mathcal{N}) \cup \mathcal{N}$  is the set of component identifiers, which can be concept and component name pairs that identify a concept component, or a component name that identifies an application component. We use  $c.n$  to denote a pair of concept name  $c \in \mathcal{T}$  and component name  $n \in \mathcal{N}$ .
- $\mathbb{C}$  is the set of component instance configurations. We use  $\mathbb{AC}$  and  $\mathbb{CC}$  to refer to the set of application and concept component instance configurations respectively ( $\mathbb{C} = \mathbb{AC} \dot{\cup} \mathbb{CC}$ ).
- $\mathbb{B}$  is the set of property bindings.

- $\mathbb{A}$  is the set of application instance configurations.

We define property bindings, component instance configurations, and application instance configurations below.

### Property Binding

A property binding binds an input property to an expression. Since the expression language and its evaluation is not an essential aspect of the semantics of Déjà Vu, we assume the existence of an expression syntax  $\mathcal{E}$  that ranges over a set of property names  $\mathcal{P}$ , and an evaluation function  $\llbracket e \rrbracket_\Gamma$  that can evaluate an expression  $e \in \mathcal{E}$  under a context  $\Gamma$  and return a value  $v \in \mathcal{V}$ . The context  $\Gamma : \mathcal{P} \rightarrow \mathcal{V}$  is a partial function mapping property names to values.

**Definition 6.2.1.** A *property binding*  $p \leftarrow e$  binds an input property  $p \in \mathcal{P}$  to an expression  $e \in \mathcal{E}$ . The expression  $e$  is an expression over a set of property names  $P_e \subseteq \mathcal{P}$ .

**Example.** A property binding `post.create.id`  $\leftarrow$  `dv.gen-id.id` binds the value of the `post.create.id` input property to the value of the output property `dv.gen-id.id` (`post.create.id`  $\in \mathcal{P}_{in}$ , `dv.gen-id.id`  $\in \mathcal{P}_{out}$ ). For simplicity, the structure in property names is not represented in the semantics. The implication of this binding is that when the value of `dv.gen-id.id` changes, the value of `post.create.id` is updated as well. How this happens is explained later.

### Component Instance Configuration

**Definition 6.2.2.** A *component instance configuration* is a tuple  $\langle \sigma, \mathcal{C}, B, \iota \rangle \in \mathbb{C}$ , where:

- $\sigma$  is the local state of the component instance. We model the local state as a partial function  $\sigma : \mathcal{P} \rightarrow \mathcal{V}$ , mapping property names to values. The local state of the component instance consists of input and output properties and, in the case of concept components, it may include private properties as well.

- $\mathcal{C} \subseteq \mathbb{C}$  is the set of children components of the component.
- $B \subseteq \mathbb{B}$  is a set of property bindings.
- $\iota \in \mathcal{I}$  is the component identifier. An application component instance configuration  $c \in \mathbb{AC}$  is a component instance configuration with  $\iota \in \mathcal{N}$ . A concept component instance configuration  $c \in \mathbb{CC}$  is a component instance configuration with  $\iota \in \mathcal{T} \times \mathcal{N}$ .

The bindings of a component instance configuration must satisfy the following properties, which prevent cycles in property bindings and prevent a component binding from accessing a property that is out of the component scope:

- All input property binding expressions in  $B$  must range over a set of output properties of child component instance configurations and input properties of the component instance configuration. Formally, if  $\mathcal{C} = \{\prec \sigma_1, \mathcal{C}_1, B_1, \iota_1 \succ, \dots, \prec \sigma_n, \mathcal{C}_n, B_n, \iota_n \succ\}$  and  $\mathcal{P}_e$  is the set of property names an expression  $e$  refers to, then  $\forall p \leftarrow e \in B$ , we have

$$p \in \mathcal{P}_{in} \cap (\text{dom } \sigma_1 \dot{\cup} \dots \dot{\cup} \text{dom } \sigma_n) \implies \mathcal{P}_e \subseteq (\mathcal{P}_{out} \cap (\text{dom } \sigma_1 \dot{\cup} \dots \dot{\cup} \text{dom } \sigma_n)) \cup (\mathcal{P}_{in} \cap \text{dom } \sigma)$$

- All output property binding expressions in  $B$  are over a set of output properties of child component instance configurations. Formally, if  $\mathcal{C} = \{\prec \sigma_1, \mathcal{C}_1, B_1, \iota_1 \succ, \dots, \prec \sigma_n, \mathcal{C}_n, B_n, \iota_n \succ\}$  and  $\mathcal{P}_e$  is the set of property names an expression  $e$  refers to, then  $\forall p \leftarrow e \in B$ , we have

$$p \in \mathcal{P}_{out} \cap \text{dom } \sigma \implies \mathcal{P}_e \subseteq \mathcal{P}_{out} \cap (\text{dom } \sigma_1 \dot{\cup} \dots \dot{\cup} \text{dom } \sigma_n)$$

Also, application component configurations cannot have private properties:

$$\iota \in \mathcal{N} \implies \mathcal{P}_{private} \cap \text{dom } \sigma = \emptyset$$

**Example.** A component instance configuration

$$\begin{aligned}
& \prec \emptyset, \\
& \left\{ \begin{array}{l} \prec \{dv.gen-id.id \mapsto 8afc\}, \emptyset, \emptyset, dv.gen-id \succ \\ \prec \{post.create.id \mapsto 8afc\}, \emptyset, \emptyset, post.create \succ \end{array} \right\}, \\
& \{post.create.id \leftarrow dv.gen-id.id\}, \\
& submit \succ
\end{aligned} \tag{6.1}$$

represents the state of an application component instance of ID `submit` with no input or output properties, a child component `create` of the `post` concept and a child component `gen-id` of `dv`, and a binding that binds the `id` input of `create` to the `id` output of `gen-id`. Both `dv.gen-id` and `post.create` have an `id` input set to value `8afc` and have no bindings or children.

## Application Instance Configuration

**Definition 6.2.3.** An *application instance configuration* is a tuple  $c; \Sigma \in \mathbb{A}$ , where:

- $c \in \mathbb{C}$  is a component instance configuration. The component instance configuration  $c$  represents the component currently executing on the client. On the first iteration of the semantics, we only allow one client.
- $\Sigma$  is the server state of the application. We model the server state  $\Sigma$  as a partial function  $\Sigma : \mathcal{T} \rightarrow \mathcal{S}$  mapping concept names to concept server states.

**Example.** A state

$$\Sigma = \left\{ \begin{array}{l} post \mapsto \{posts \mapsto \{(id : 8afc, content : "hello")\}\} \\ dv \mapsto \emptyset \end{array} \right\} \tag{6.2}$$

represents the server state of an application, where the server state for the concept `post` has a `posts` property with a singleton value containing a post record `(id : 8afc, content : "hello")`, and the server state for `dv` is empty.

An application instance configuration  $c; \Sigma$ , where  $c$  is equal to the component instance configuration of (6.1) and  $\Sigma$  is equal to the concept server state defined in (6.2), represents an application instance configuration with component instance configuration  $c$  and a global application state  $\Sigma$ . The server state that the child concept component `post.create` of  $c$  can read/write is  $\Sigma(\text{post})$ , and the server state the child concept component `dv.gen-id` of  $c$  can read/write is  $\Sigma(\text{dv})$ .

## Update Function

We define a function `update` that propagates data updates across the component tree according to the bindings. The function updates the input properties of components in the tree by re-evaluating the bindings under the component context. It is defined as follows:

$$\begin{aligned} & \text{update}(\prec \sigma, \{\prec \sigma_1, \mathcal{C}_1, B_1, \iota_1 \succ, \dots, \prec \sigma_n, \mathcal{C}_n, B_n, \iota_n \succ\}, B, \iota \succ) = \\ & \text{let } V = \llbracket B \rrbracket_{\sigma \dot{\cup} \sigma_1 \dot{\cup} \dots \dot{\cup} \sigma_n} \text{ in} \\ & \quad \prec \sigma \otimes V, \\ & \quad \{\text{update}(\prec \sigma_1 \otimes V, \mathcal{C}_1, B_1, \iota_1 \succ), \dots, \text{update}(\prec \sigma_n \otimes V, \mathcal{C}_n, B_n, \iota_n \succ)\}, \\ & \quad B, \iota \succ \end{aligned}$$

The store update operator  $\otimes$  updates the property values in the first store with the values in the second store:

$$(f \otimes g)(x) = \begin{cases} f(x) & \text{if } x \in \text{dom } f \wedge x \notin \text{dom } g \\ g(x) & \text{if } x \in \text{dom } f \wedge x \in \text{dom } g \end{cases}$$

The `update` function takes a component instance configuration and recursively updates all the input properties of the component children. It updates all the input properties by updating the property values with the values in the map  $V = \llbracket B \rrbracket_{\sigma \dot{\cup} \sigma_1 \dot{\cup} \dots \dot{\cup} \sigma_n}$ . The map  $V$  is the result of evaluating property bindings  $B$  under a context  $\sigma \dot{\cup} \sigma_1 \dot{\cup} \dots \dot{\cup} \sigma_n$ , which includes the state of the current component ( $\sigma$ ) and the state of all the immediate children of the current component ( $\sigma_1, \dots, \sigma_n$ ). The `update` function uses the expression evaluation function  $\llbracket e \rrbracket_{\Gamma}$ , which we lift to operate

over bindings. That is,  $\llbracket B \rrbracket_\Gamma$  evaluates  $e$  under context  $\Gamma$  for each  $p \leftarrow e \in B$ , and returns a map of property names to values ( $\text{dom } B \rightarrow \mathcal{V}$ ).

Because there are no cycles in the bindings and because the output properties of a component can't refer to its own inputs, the **update** function can update an entire component tree in one pass and all property values will be in a consistent state with respect to the bindings.

## 6.2.2 Concept Component Behavior

The transition relation  $\longrightarrow \subseteq \mathbb{A} \times \mathbb{A}$  includes a set of built-in transitions  $\mathbb{A} \times \mathbb{A}$  that determine the behavior of concept components, which are implemented in a different language and are not included in the semantics. The transitions that determine the behavior of concept components are all of the form:

$$\prec \sigma, \mathcal{C}, B, c.n \succ; \Sigma \dot{\cup} \{c \mapsto S_c\} \longrightarrow \prec \sigma', \mathcal{C}', B', c.n \succ; \Sigma \dot{\cup} \{c \mapsto S'_c\}$$

The concept transitions take an application instance configuration with a concept component instance configuration of ID  $c.n$ , state  $\sigma$ , children  $\mathcal{C}$ , and bindings  $B$  to a new application instance configuration with a new component instance configuration with the same ID as the one before, but with a new state  $\sigma'$ , children  $\mathcal{C}'$  and bindings  $B'$ . The rule also allows the component to update the global state of the application by mutating the concept server state from a state  $S_c \in \mathcal{S}$  to  $S'_c \in \mathcal{S}$ . The concept transitions encapsulate concept component behavior such as updating an output property when the user selects an option in a dropdown, or saving a new post to the database when the user clicks on a button.

**Example.** The behavior of the `dv.gen-id` concept component is given by the following set of transitions, where `ID` is the set of valid unique identifiers:

$$\{ \prec \emptyset, \emptyset, \emptyset, \text{dv.gen-id} \succ; \Sigma \longrightarrow \prec \{\text{dv.gen-id.id} \mapsto i\}, \emptyset, \emptyset, \text{dv.gen-id} \succ; \Sigma \mid i \in \text{ID} \}$$



$$\text{CHILD-STEP} \frac{\prec \sigma_i, \mathcal{C}_i, B_i, \iota_i \succ; \Sigma \longrightarrow \prec \sigma'_i, \mathcal{C}'_i, B'_i, \iota_i \succ; \Sigma'}{\prec \sigma, \mathcal{C} \dot{\cup} \{\prec \sigma_i, \mathcal{C}_i, B_i, \iota_i \succ\}, B, n \succ; \Sigma \longrightarrow \text{update}(\prec \sigma, \mathcal{C} \dot{\cup} \{\prec \sigma'_i, \mathcal{C}'_i, B'_i, \iota_i \succ\}, B, n \succ); \Sigma'}$$

Figure 6-1: Inference rule for application component instance configurations

The transition states that the `dv.gen-id` component can, starting from an empty local state, output an ID value  $i \in \text{ID}$ . The `gen-id` component outputs an ID value  $i$  by adding to its local state a new property `dv.gen-id.id` with value  $i$ . The global application state  $\Sigma$  remains unchanged.

While nothing in the rule prevents the same ID value from being generated more than once, in practice the implementation generates random IDs with a very low probability of collision.

### 6.2.3 Rules

On the first iteration of the semantics, we only have one inference rule CHILD-STEP, which is shown in Fig. 6-1. CHILD-STEP defines the behavior of an application instance configuration in terms of the behavior of a child of the application component configuration in the application instance configuration. The child can be a concept component instance configuration or an application component instance configuration. We use  $\iota_i$  for the ID of the child to indicate that the child can be a concept or application component instance configuration and assume  $\iota_i \in \mathcal{I}$ . We use  $n$  to indicate the component instance configuration in the conclusion of the inference rule must be an application instance configuration, and assume  $n \in \mathcal{N} \subseteq \mathcal{I}$ .

An application component that has no children has no behavior at all. This is because the only behavior of an application component is to coordinate the behavior of its children components.

### 6.2.4 Initial Application Instance Configuration

The initial configuration of a concept component and server state are given by two concept catalog functions. The function  $\Theta_{\text{CC}} : \mathcal{T} \times \mathcal{N} \rightarrow \mathbb{C}\mathbb{C}$  takes a concept component identifier  $c.n \in \mathcal{T} \times \mathcal{N}$  and returns an initial concept component instance

configuration  $cc \in \mathbb{CC}$ . The function  $\Theta_S : \mathcal{T} \rightarrow \mathcal{S}$  takes a concept name  $c \in \mathcal{T}$  and returns an initial concept state  $S_0 \in \mathcal{S}$ . The initial global application state is  $\Sigma^0 = \{c \mapsto \Theta_S(c) \mid c \in \mathbf{IC}\}$  where  $\mathbf{IC} \subseteq \mathcal{T}$  is the set of concepts used in the application. We formalize the translation from a core syntax of Déjà Vu to the semantics elements defined in the operational semantics later in §6.5.

## Example: Submit Post

Let's consider a small example application with the following pseudocode:

```

1 | submit := [ // define new application component submit
2 |   dv.gen-id // include the built-in gen-id component
3 |   // include the create concept component of post and bind its id input
4 |   post.create  id = dv.gen-id.id
5 | ]

```

The application consists of one application component called `submit`. The `submit` application component (lines 1-5) includes the `gen-id` concept component of `dv` (line 2) and the `create` component of the `post` concept (line 4). We also bind the `id` input of `create` to the `id` output of `gen-id` (line 4). In this example, we are going to start with a fresh instance of `submit`, have `dv.gen-id` generate a unique ID, and have `post.create` save a new post to the database with the ID generated by `dv.gen-id`.

## Initial Application Instance Configuration

**Initial Concept Configurations.** The initial component instance configuration for `dv.gen-id` and `post.create` is  $\Theta_{\mathbb{CC}}(\text{dv.gen-id}) = \prec \emptyset, \emptyset, \emptyset, \text{dv.gen-id} \succ$  and  $\Theta_{\mathbb{CC}}(\text{post}) = \prec \{\text{post.create.id} \mapsto \perp\}, \emptyset, \emptyset, \text{post.create} \succ$  respectively. Both components have no bindings or children. The `dv` component starts with an empty local state, and `post` starts with an `id` input property set to  $\perp$ . The initial concept

server state for `post` is  $\Theta_S(\text{post}) = \{\text{posts} \mapsto \emptyset\}$  and the initial state for `dv` is  $\Theta_S(\text{dv}) = \emptyset$ .

**Children.** The `submit` component has two children: `dv.gen-id` and `post.create`. The initial configuration for the concept components is determined by  $\Theta_{CC}$ . The initial set of children components of `submit` is:

$$C^0 = \left\{ \begin{array}{l} \prec \emptyset, \emptyset, \emptyset, \text{dv.gen-id} \succ \\ \prec \{\text{post.create.id} \mapsto \perp\}, \emptyset, \emptyset, \text{post.create} \succ \end{array} \right\} \quad (6.3)$$

**Bindings.** In `submit`, there's only one property binding, which binds the `id` property of the `create` component of `post` with the expression `dv.gen-id.id`. In the set of bindings for `submit`, we fully-qualify the `id` input, so that the binding on line 4 becomes `post.create.id`  $\leftarrow$  `dv.gen-id.id` and the set of bindings for `submit` is:

$$B = \{\text{post.create.id} \leftarrow \text{dv.gen-id.id}\} \quad (6.4)$$

**Initial Application Instance Configuration.** Since `submit` has no inputs, it starts with an empty local state  $\sigma = \emptyset$ . The initial component instance configuration for `submit` is

$$\text{submit}^0 = \prec \emptyset, C^0, B, \text{submit} \succ \quad (6.5)$$

where  $C^0$  and  $B$  are defined in (6.3) and (6.4) respectively. The initial application state  $\Sigma^0$  is

$$\Sigma^0 = \left\{ \begin{array}{l} \text{post} \mapsto \{\text{posts} \mapsto \emptyset\} \\ \text{dv} \mapsto \emptyset \end{array} \right\} \quad (6.6)$$

and the initial application instance configuration for the example application is

$$\prec \emptyset, C^0, B, \text{submit} \succ; \Sigma^0$$

## Concept Component Behavior

**Component `dv.gen-id`.** To explain the behavior of `submit`, we must first give the behavior of the concept components `dv.gen-id` and `post.create`. As explained in §6.2.2, the behavior of `dv.gen-id` is given by the following set of transitions, where `ID` is the set of valid unique identifiers:

$$\{ \prec \emptyset, \emptyset, \emptyset, \text{dv.gen-id} \succ; \Sigma \longrightarrow \prec \{ \text{dv.gen-id.id} \mapsto i \}, \emptyset, \emptyset, \text{dv.gen-id} \succ; \Sigma \mid i \in \text{ID} \}$$

**Component `post.create`.** The behavior of `post.create` is given by the following set of transitions, where `Content` is the set of valid post contents:

$$\begin{aligned} & \{ \\ & \prec \{ \text{post.create.id} \mapsto p_{id} \}, \emptyset, \emptyset, \text{post.create} \succ; \\ & \Sigma \dot{\cup} \{ \text{post} \mapsto \{ \text{posts} \mapsto \{ (\text{id}: id_1, \text{content}: c_1), \dots, (\text{id}: id_n, \text{content}: c_n) \} \} \} \\ & \longrightarrow \\ & \prec \{ \text{post.create.id} \mapsto p_{id} \}, \emptyset, \emptyset, \text{post.create} \succ; \\ & \Sigma \dot{\cup} \{ \text{post} \mapsto \{ \\ & \quad \text{posts} \mapsto \{ (\text{id}: id_1, \text{content}: c_1), \dots, (\text{id}: id_n, \text{content}: c_n) \} \} \dot{\cup} \\ & \quad \{ (\text{id}: p_{id}, \text{content}: p_c) \} \} \\ & \mid p_{id} \in \text{ID} \wedge p_{id} \notin \{ id_1, \dots, id_n \}, p_c \in \text{Content} \\ & \} \end{aligned}$$

The transition says that the `post.create` component can, starting from a local state in which the `id` input is `pid`, add a new post to `posts` with ID `pid` and any content `pc ∈ Content`, as long as there's no other post with the same ID already in `posts`.

## Example Derivation

In this example derivation, we are going to have the application instance do two steps. On the first step, the `gen-id` component outputs a unique ID value `8afc` for the post. On the second step, `create` saves a new post to the database, using the generated ID as the post ID. Each step has a proof justifying the step.

1. **Output ID.** In the first step, we apply the CHILD-STEP rule:

$$\text{CHILD-STEP} \frac{\begin{array}{c} \prec \emptyset, \emptyset, \emptyset, \text{dv.gen-id} \succ; \Sigma^0 \longrightarrow \\ \prec \{\text{dv.gen-id.id} \mapsto \text{8afc}\}, \emptyset, \emptyset, \text{dv.gen-id} \succ; \Sigma^0 \end{array}}{\prec \emptyset, C^0, B, \text{submit} \succ; \Sigma^0 \longrightarrow \prec \emptyset, C^1, B, \text{submit} \succ; \Sigma^0}$$

where `8afc`  $\in$  ID.  $C^0$ ,  $B$ , and  $\Sigma^0$  were defined previously in (6.3), (6.4), and (6.6) respectively. The new children set  $C^1$  is defined as follows:

$$C^1 = \left\{ \begin{array}{l} \prec \{\text{dv.gen-id.id} \mapsto \text{8afc}\}, \emptyset, \emptyset, \text{dv.gen-id} \succ \\ \prec \{\text{post.create.id} \mapsto \text{8afc}\}, \emptyset, \emptyset, \text{post.create} \succ \end{array} \right\}$$

The updated component with children set  $C^1$  is the result of running

$$\text{update}(\prec \emptyset, \left\{ \begin{array}{l} \prec \{\text{dv.gen-id.id} \mapsto \text{8afc}\}, \emptyset, \emptyset, \text{dv.gen-id} \succ \\ \prec \{\text{post.create.id} \mapsto \perp\}, \emptyset, \emptyset, \text{post.create} \succ \end{array} \right\}, B, \text{submit} \succ)$$

where

$$\llbracket p \leftarrow \text{dv.gen-id.id} \rrbracket_{\Gamma} = \{p \mapsto e_{id}\} \quad \text{if } \Gamma(\text{dv.gen-id.id}) = e_{id}$$

2. **Save Post.** In the second and final step, we apply the same CHILD-STEP rule, but this time with the `post.create` child taking a step to save a new post to the database:

$$\text{CHILD-STEP} \frac{\begin{array}{l} \prec \{\text{post.create.id} \mapsto 8\text{afc}\}, \emptyset, \emptyset, \text{post.create} \succ; \Sigma^0 \longrightarrow \\ \prec \{\text{post.create.id} \mapsto 8\text{afc}\}, \emptyset, \emptyset, \text{post.create} \succ; \Sigma^1 \end{array}}{\prec \emptyset, \mathcal{C}^1, B, \text{submit} \succ; \Sigma^0 \longrightarrow \prec \emptyset, \mathcal{C}^1, B, \text{submit} \succ; \Sigma^1}$$

where

$$\Sigma^1 = \left\{ \begin{array}{l} \text{post} \mapsto \{\text{posts} \mapsto \{(\text{id} : 8\text{afc}, \text{content} : \text{"hello"})\}\} \\ \text{dv} \mapsto \emptyset \end{array} \right\}$$

and "hello"  $\in$  Content.

The result is that `post.create` has saved a new post with the ID given by `dv.gen-id`.

## 6.3 Second Iteration: Clients

In the second iteration, we add clients and navigation. Instead of assuming there's only one client in the application instance configuration, we will now define clients and update the definition of an application instance configuration to contain a set of clients (§6.3.1). Because we update the definition of an application instance configuration, we also update the form of the concept component built-in transitions (§6.3.2). We also add new rules so that new clients can be added to the application instance and existing clients can be removed, and we also allow a concept component to cause a client to navigate to another component (§6.3.3).

### 6.3.1 Definitions

**Definition 6.3.1.** A *client* is a tuple  $\langle c, x \rangle$ , where:

- $c \in \mathbb{C}$  is a component instance configuration. The component  $c$  represents the component currently executing on the client.

- $x \in \mathbb{A}\mathbb{C}_0 \cup \{\perp\}$  is an application component instance configuration to navigate to, or  $\perp$  if there's no navigation request. The set  $\mathbb{A}\mathbb{C}_0 \subseteq \mathbb{A}\mathbb{C}$  is the set of initial application component instance configurations.

Our client structure is essentially a continuation [55], because it represents the control flow of the program running on the client. Like in languages with first-class continuations, our client structure allows a concept component to access and change the control flow of a client by updating the client structure to have a navigation request.

**Definition 6.3.2.** An *application instance configuration* is a tuple  $\Lambda; \Sigma$ , where:

- $\Lambda$  is a set of clients.
- $\Sigma$  is the server state of the application. Like in the previous iteration of the semantics, we model the server state  $\Sigma$  as a partial function  $\Sigma : \mathcal{T} \rightarrow \mathcal{S}$  mapping concept names to concept server states.

### 6.3.2 Concept Component Behavior

Since we have updated the definition of an application instance configuration, we have to update the transitions that determine the behavior of concepts components. The concept component transitions now have the form:

$$L \dot{\cup} \{\langle \prec \sigma, \mathcal{C}, B, c.n \succ, \perp \rangle\}; \Sigma \dot{\cup} \{c \mapsto S_c\} \longrightarrow \\ L \dot{\cup} \{\langle \prec \sigma', \mathcal{C}', B', c.n \succ, x \rangle\}; \Sigma \dot{\cup} \{c \mapsto S'_c\}$$

The concept transitions take an application instance configuration with a client on a concept instance configuration of ID  $c.n$ , state  $\sigma$ , children  $\mathcal{C}$ , and bindings  $B$  to a new application instance configuration with a new component instance configuration with the same ID as the one before, but with a new state  $\sigma'$ , children  $\mathcal{C}'$  and bindings  $B'$ . The rule also allows the component to update the global state of the application by mutating the concept server state from a state  $S_c \in \mathcal{S}$  to  $S'_c \in \mathcal{S}$  and to add a navigation request  $x \in \mathbb{A}\mathbb{C}_0$  to the client. Concept components can't make a transition if there's an active navigation request on the client.

$$\begin{array}{l}
\text{CHILD-STEP} \frac{L \dot{\cup} \{ \langle \prec \sigma_i, \mathcal{C}_i, B_i, \iota_i \succ, \perp \rangle \}; \Sigma \longrightarrow L \dot{\cup} \{ \langle \prec \sigma'_i, \mathcal{C}'_i, B'_i, \iota_i \succ, x \rangle \}; \Sigma'}{L \dot{\cup} \{ \langle \prec \sigma, \mathcal{C} \dot{\cup} \{ \langle \prec \sigma_i, \mathcal{C}_i, B_i, \iota_i \succ \}, B, n \succ, \perp \rangle \}; \Sigma \longrightarrow \\
L \dot{\cup} \{ \langle \text{update}(\prec \sigma, \mathcal{C} \dot{\cup} \{ \langle \prec \sigma'_i, \mathcal{C}'_i, B'_i, \iota_i \succ \}, B, n \succ), x \rangle \}; \Sigma'} \\
\\
\text{CLIENT-NAV} \frac{c_{next} \in \mathbb{A}\mathbb{C}_0}{L \dot{\cup} \{ \langle c, c_{next} \rangle \}; \Sigma \longrightarrow L \dot{\cup} \{ \langle c_{next}, \perp \rangle \}; \Sigma} \\
\\
\text{CLIENT-ADD} \frac{c \in \mathbb{A}\mathbb{C}_0}{L; \Sigma \longrightarrow L \dot{\cup} \{ \langle c, \perp \rangle \}; \Sigma} \\
\\
\text{CLIENT-DROP} \frac{}{L \dot{\cup} \{ l \}; \Sigma \longrightarrow L; \Sigma}
\end{array}$$

Figure 6-2: Inference rules for application instance configurations with clients

### 6.3.3 Rules

In this iteration, we add new rules for navigation, and for adding and removing clients to an application instance configuration. We also update the original CHILD-STEP rule to operate on only one of the clients at a time. The rules for the second iteration of the semantics are shown in Fig. 6-2.

The CHILD-STEP rule only applies to a client when the client has no active navigation request. If the client has an active navigation request, the client is blocked until an application of the CLIENT-NAV rule causes the client to navigate to the next component and removes the navigation request. New clients can be added to an application instance configuration at any time with the CLIENT-ADD rule. Existing clients can be removed from an application instance configuration at any time with the CLIENT-DROP rule.

#### **Example: Link to Submit Post**

In this example, we extend the previous example application with a new component `home` that includes a link to `submit`:



```

1 | submit := [ // define new application component submit
2 |   dv.gen-id // include the built-in gen-id component
3 |   // include the create concept component of post and bind its id input
4 |   post.create id = dv.gen-id.id
5 | ]
6 | home := [ // define new application component home
7 |   dv.link href = submit // include a link to submit
8 | ]

```

In this example, we are going to start with an application instance that has no clients. Then we are going to add a new client with the `home` component and have `dv.link` cause a navigation to `submit`.

### Initial Application Instance Configuration

The set  $\mathbb{AC}_0$  includes the initial component instance configuration for `submit` and for `home`. The initial configuration for `submit`,  $\text{submit}^0$ , is the same as in (6.5). The initial configuration for `home` is

$$\begin{aligned}
\text{home}^0 = & \\
& \langle \emptyset, \{ \langle \{\text{href} \mapsto \perp\}, \emptyset, \emptyset, \text{dv.link} \rangle \}, \\
& \{\text{dv.link.href} \leftarrow \text{submit}^0\}, \text{home} \rangle
\end{aligned} \tag{6.7}$$

The initial set of clients is empty and the initial global application state  $\Sigma^0$  is the same as in (6.6), because the set of included concepts is the same as in the previous example. The initial application instance configuration is  $\emptyset; \Sigma^0$  and  $\mathbb{AC}_0 = \{\text{home}^0, \text{submit}^0\}$ .

### Link Behavior

The behavior of `dv.link` is given by the following set of transitions:

$$\begin{aligned} & \{L \dot{\cup} \{\langle \prec \{\text{href} \mapsto c\}, \emptyset, \emptyset, \text{dv.link } \succ, \perp \rangle; \Sigma\} \longrightarrow \\ & L \dot{\cup} \{\langle \prec \{\text{href} \mapsto c\}, \emptyset, \emptyset, \text{dv.link } \succ, c \rangle; \Sigma\} \mid c \in \mathbb{A}\mathbb{C}_0 \} \end{aligned}$$

A `dv.link` component with an `href` input equal to a component  $c \in \mathbb{A}\mathbb{C}_0$  can set the client navigation request to  $c$  if there's no active navigation request on the client.

### Example Derivation

In this example derivation, we are going to have the application instance do three steps. On the first step, a new client with the `home` component is added to the application instance. On the second step, `dv.link` sets a navigation request to `submit` on the client. On the third step, the client navigates to `submit`. Each step has a proof justifying the step. We are going to use `home0`, defined in (6.7), to refer to the initial configuration for the `home` component. And we are going to use `submit0`, defined in (6.5), to refer to the initial configuration for the `submit` component.

1. **New Client.** In the first step, we apply the CLIENT-ADD rule:

$$\text{CLIENT-ADD} \frac{\text{home}^0 \in \mathbb{A}\mathbb{C}_0}{\emptyset; \Sigma^0 \longrightarrow \{\langle \text{home}^0, \perp \rangle\}; \Sigma^0}$$

The application of this rule adds a new client to the application instance configuration. The new client runs `home0` and has no active navigation request.

2. **Link Adds Navigation Request to submit.** In the second step, we apply the CHILD-STEP rule:

$$\text{CHILD-STEP} \frac{\begin{array}{l} \{\langle \prec \{\text{href} \mapsto \text{submit}^0\}, \emptyset, \emptyset, \text{dv.link } \succ, \perp \rangle; \Sigma^0 \longrightarrow \\ \{\langle \prec \{\text{href} \mapsto \text{submit}^0\}, \emptyset, \emptyset, \text{dv.link } \succ, \text{submit}^0 \rangle; \Sigma^0 \end{array}}{\{\langle \text{home}^0, \perp \rangle\}; \Sigma^0 \longrightarrow \{\langle \text{home}^0, \text{submit}^0 \rangle\}; \Sigma^0}$$

The application of this rule updates the client configuration to have a navigation request to `submit0`, which is the value of the `href` input of `dv.link`.

3. **Client Navigates to submit.** In the third and last step, we apply the CLIENT-NAV rule to cause the client to navigate to `submit`:

$$\text{CLIENT-NAV} \frac{\text{submit}^0 \in \mathbb{A}\mathbb{C}_0}{\{\langle \text{home}^0, \text{submit}^0 \rangle\}; \Sigma^0 \longrightarrow \{\langle \text{submit}^0, \perp \rangle\}; \Sigma^0}$$

Note that at this point the application instance configuration  $\langle \text{submit}^0, \perp \rangle; \Sigma^0$  is the same as the initial configuration of the previous example, but augmented with the notion of a client. Using the same steps as in the previous example, we could have the client now save a new post to the database.

## 6.4 Third Iteration: Full Semantics

In the third and final iteration of the semantics we add transaction components. We update the set of component identifiers to add a type to application component identifiers. The type can be  $\vee$ , which represents a regular or “or” component, or  $\wedge$ , which represents a transaction or “and” component (§6.4.1). We also distinguish between an operation transition, that is never synchronized, and an action transition, that may be synchronized or not depending on the type of the application component.

Before this iteration, transitions were unlabelled, but in this iteration we add transition labels. A transition with label *op* represents an operation transition, which is never synchronized. A transition with label *a* represents an action transition, which may be synchronized or not depending on the component type. For simplicity, we don’t distinguish between *eval* and *exec* actions in the semantics. Both *eval* and *exec* are synchronized in the same way, so it is easy to extend the semantics to distinguish between *eval/exec* actions: instead of a label *a*, have labels *eval* and *exec* and change each rule with an action transition into two rules that are the same but one has an *eval* transition and the other an *exec* transition replacing the original action transition.

The transition relation  $\longrightarrow \subseteq \mathbb{A} \times \{op, a\} \times \mathbb{A}$  includes a set of built-in transitions that determine the behavior of concept components (§6.4.2) and rules for synchronizing actions depending on the component type (§6.4.3).

### 6.4.1 Definitions

The definition of a component instance configuration is the same as before, but we redefine the set of component identifiers to  $\mathcal{I} \subseteq (\mathcal{T} \times \mathcal{N}) \cup (\mathcal{N} \times \{\wedge, \vee\})$ .

### 6.4.2 Concept Component Behavior

In this iteration, concept components can take an action step or an operation step. The action transitions are all of the form:

$$\begin{aligned} L \dot{\cup} \{\langle \prec \sigma, \mathcal{C}, B, c.n \succ, \perp \rangle\}; \Sigma \dot{\cup} \{c \mapsto S_c\} &\xrightarrow{a} \\ L \dot{\cup} \{\langle \prec \sigma', \mathcal{C}', B', c.n \succ, x \rangle\}; \Sigma \dot{\cup} \{c \mapsto S'_c\} & \end{aligned}$$

where  $S_c, S'_c \in \mathcal{S}$ . In an action transition, concept components can modify the concept server state. The operation transitions are all of the form:

$$L \dot{\cup} \{\langle \prec \sigma, \mathcal{C}, B, c.n \succ, \perp \rangle\}; \Sigma \xrightarrow{op} L \dot{\cup} \{\langle \prec \sigma', \mathcal{C}', B', c.n \succ, x \rangle\}; \Sigma$$

An operation step cannot modify the global state, which is why the global state  $\Sigma$  remains unchanged after a concept component takes an operation step. The reason for this restriction is to have all concept transitions that can produce a side-effect on the server be an action, which enables the user to synchronize the action with other actions. If otherwise, there could be concept components that fetch data from their server but because the fetch doesn't happen as part of an action there's no way for the developer to, for example, synchronize the action with `authenticate` to prevent a malicious user from seeing information they are not supposed to see. We could say that all concept transitions are action transitions, but it is useful to allow concept components to take a step without triggering all their sibling concept components.

For example, a component with a dropdown might update an output property with the selected item value every time the user selects a new item. Since this is a step that doesn't involve reading or writing the server state, it could be an operation.

### 6.4.3 Rules

In this iteration, we add new rules for synchronizing actions depending on whether the component is a transaction component or not. The rules for the third iteration of the semantics are shown in Fig. 6-3.

In this iteration, we replace CHILD-STEP with three new rules: CHILD-OP, CHILD-OR and CHILD-AND. CHILD-OP represents a step that is not synchronized. In CHILD-OR, a child concept component is taking an action step, but because the application component that contains the concept component is an “or” component, the action step is not synchronized with the other concept components. On the other hand, in CHILD-AND, if a child concept component takes an action step, all other concept components in the containing application component must take an action step as well. This is because the containing application component is a transaction or “and” component.

In CHILD-AND, all child concept components must take an action step together if one of them does. This restriction is given by the requirement  $\mathcal{C} \subseteq \mathbb{A}\mathbb{C}$  in the CHILD-AND rule. Since we ask that all the components in the transaction must be from different concepts, which is given by the requirement  $c_i \neq c_j \forall i, j \in 1 \dots m$ , we know that all the updated application states  $\Sigma'_1, \dots, \Sigma'_m$  have updated different concept server states of the global application state. It is therefore safe to compute the new global application state by taking the previous store  $\Sigma$  and using the store update operator  $\otimes$  to update its property values with  $\Sigma'_1, \dots, \Sigma'_m$ , regardless of the order in which  $\otimes$  is applied to the  $\Sigma'_1, \dots, \Sigma'_m$ . It is important, however, for  $\Sigma$  to go first, so as to not override a new concept server state with an old value.

$$\begin{array}{c}
\text{CHILD-OP} \\
\hline
L\dot{\cup}\{\langle\langle\sigma_i, C_i, B_i, \iota_i \succ, \perp\rangle\rangle; \Sigma \xrightarrow{op} L\dot{\cup}\{\langle\langle\sigma'_i, C'_i, B'_i, \iota_i \succ, x\rangle\rangle; \Sigma'\} \\
L\dot{\cup}\{\langle\langle\sigma, C\dot{\cup}\{\langle\sigma_i, C_i, B_i, \iota_i \succ\rangle, B, m \succ, \perp\rangle\}; \Sigma \xrightarrow{op} \rightarrow \\
L\dot{\cup}\{\langle\text{update}(\langle\sigma, C\dot{\cup}\{\langle\sigma'_i, C'_i, B'_i, \iota_i \succ\rangle\}), B, m \succ\rangle, x\rangle\}; \Sigma'\} \\
\hline
\text{CHILD-OR} \\
\hline
L\dot{\cup}\{\langle\langle\sigma_i, C_i, B_i, c.n \succ, \perp\rangle\rangle; \Sigma \xrightarrow{a} L\dot{\cup}\{\langle\langle\sigma'_i, C'_i, B'_i, c.n \succ, x\rangle\rangle; \Sigma'\} \\
L\dot{\cup}\{\langle\langle\sigma, C\dot{\cup}\{\langle\sigma_i, C_i, B_i, c.n \succ\rangle, B, m_\vee \succ, \perp\rangle\}; \Sigma \xrightarrow{op} \rightarrow \\
L\dot{\cup}\{\langle\text{update}(\langle\sigma, C\dot{\cup}\{\langle\sigma'_i, C'_i, B'_i, c.n \succ\rangle\}), B, m_\vee \succ\rangle, x\rangle\}; \Sigma'\} \\
\hline
L\dot{\cup}\{\langle\langle\sigma_1, C_1, B_1, c_1.n_1 \succ, \perp\rangle\rangle; \Sigma \xrightarrow{a} L\dot{\cup}\{\langle\langle\sigma'_1, C'_1, B'_1, c_1.n_1 \succ, x_1\rangle\rangle; \Sigma'_1 \\
\vdots \\
L\dot{\cup}\{\langle\langle\sigma_m, C_m, B_m, c_m.n_m \succ, \perp\rangle\rangle; \Sigma \xrightarrow{a} L\dot{\cup}\{\langle\langle\sigma'_m, C'_m, B'_m, c_m.n_m \succ, x_m\rangle\rangle; \Sigma'_m \\
\mathcal{C} \subseteq \mathbb{AC} \\
x \in \{x_1, \dots, x_m\} \setminus \{\perp\} \vee \{x_1, \dots, x_m\} = \{\perp\} = \{x\} \\
c_i \neq c_j \forall i, j \in 1 \dots m \\
\hline
\text{CHILD-AND} \\
\hline
L\dot{\cup}\{\langle\langle\sigma, C\dot{\cup}\{\langle\sigma_i, C_i, B_i, c_1.n_1 \succ, \dots, \langle\sigma_m, C_m, B_m, c_m.n_m \succ\rangle, B, n_\wedge \succ, \perp\rangle\}; \\
\Sigma \xrightarrow{op} \rightarrow \\
L\dot{\cup}\{\langle\text{update}(\langle\sigma, C\dot{\cup}\{\langle\sigma'_1, C'_1, B'_1, c_1.n_1 \succ, \dots, \langle\sigma'_m, C'_m, B'_m, c_m.n_m \succ\rangle, B, n_\wedge \succ\rangle, x\rangle\}; \\
\Sigma \otimes \Sigma'_1 \otimes \dots \otimes \Sigma'_m \\
\hline
\text{CLIENT-NAV, CLIENT-ADD, and CLIENT-DROP rules are the same as in Fig. 6-2 with all transitions labelled } op
\end{array}$$

Figure 6-3: Inference rules for application instance configurations with synchronization

## Example: Submit Post with Event

In this example, we extend the previous example application by making `submit` a transaction component and adding `create-event`:

```
1 | tx submit := [// define new transaction application component submit
2 |   dv.gen-id // include the built-in gen-id component
3 |   // include the create concept component of post and bind its id input
4 |   post.create id = dv.gen-id.id
5 |   // include the create concept component of event and bind its id input
6 |   event.create id = dv.gen-id.id
7 | ]
8 | home := [// define new application component home
9 |   dv.link href = submit // include a link to submit
10| ]
```

This application allows a user to post about interesting events happening on campus. For example, a client can post “New exhibit at the MIT Museum!” with a start date 03/03/2020 and an end date 03/05/2020.

### Initial Application Instance Behavior

Since the steps for taking an application instance configuration with no clients to one with one client on `submit` is similar to the steps in the previous example, we are going to start this example from an application instance configuration in which we already have a client on the `submit` component. Also, we are going to assume the ID value generated by `gen-id` was already propagated through the bindings to `post.create` and `event.create`.

The set of bindings for `submit` is:

$$B = \left\{ \begin{array}{l} \text{post.create.id} \leftarrow \text{dv.gen-id.id} \\ \text{event.create.id} \leftarrow \text{dv.gen-id.id} \end{array} \right\} \quad (6.8)$$

Since we assume that the initial `submit` component has already propagated the generated ID through the bindings, we have:

$$\begin{aligned} \text{dv.gen-id}^I &= \prec \{ \text{dv.gen-id.id} \mapsto 8\text{afc} \}, \emptyset, \emptyset, \text{dv.gen-id} \succ \\ \text{post.create}^I &= \prec \{ \text{post.create.id} \mapsto 8\text{afc} \}, \emptyset, \emptyset, \text{post.create} \succ \\ \text{event.create}^I &= \prec \{ \text{event.create.id} \mapsto 8\text{afc} \}, \emptyset, \emptyset, \text{event.create} \succ \end{aligned} \quad (6.9)$$

and the set of children of `submit`<sup>I</sup> is:

$$\mathcal{C} = \left\{ \begin{array}{l} \text{dv.gen-id}^I \\ \text{post.create}^I \\ \text{event.create}^I \end{array} \right\} \quad (6.10)$$

The initial server state  $\Sigma^0$  includes the state for `event`, in addition to the state for `dv` and `post`:

$$\Sigma^0 = \left\{ \begin{array}{l} \text{dv} \mapsto \emptyset \\ \text{post} \mapsto \{ \text{posts} \mapsto \emptyset \} \\ \text{event} \mapsto \{ \text{events} \mapsto \emptyset \} \end{array} \right\} \quad (6.11)$$



## Create Event Behavior

The behavior of `event.create` is given by the following set of transitions, where `Date` is a set of valid dates:

$$\begin{array}{l} \{ \\ \prec \{ \text{event.create.id} \mapsto e_{id} \}, \emptyset, \emptyset, \text{event.create} \succ; \\ \Sigma \dot{\cup} \{ \text{event} \mapsto \{ \\ \quad \text{events} \mapsto \{ (\text{id}: id_1, \text{start}: s_1, \text{end}: e_1), \dots, (\text{id}: id_n, \text{start}: s_n, \text{end}: e_n) \} \} \\ \xrightarrow{a} \\ \prec \{ \text{event.create.id} \mapsto e_{id} \}, \emptyset, \emptyset, \text{event.create} \succ; \\ \Sigma \dot{\cup} \{ \text{event} \mapsto \{ \\ \quad \text{events} \mapsto \{ (\text{id}: id_1, \text{start}: s_1, \text{end}: e_1), \dots, (\text{id}: id_n, \text{start}: s_n, \text{end}: e_n) \} \} \dot{\cup} \\ \quad \{ (\text{id}: e_{id}, \text{start}: e_{start}, \text{end}: e_{end}) \} \\ | e_{id} \in \text{ID}, e_{id} \notin \{ id_1, \dots, id_n \}, e_{start}, e_{end} \in \text{Date} \\ \} \end{array}$$

The transition says that the `create` component of `event` can, starting from a local state in which the `id` input is  $e_{id}$ , take an action step and add a new event to `events` with `id`  $e_{id} \in \text{ID}$  and dates  $e_{start}, e_{end} \in \text{Date}$ , as long as there's no other event with the same ID already in `events`.

## Example Derivation

In this example derivation, we are going to have the application instance do a step, in which both `post.create` and `event.create` take an action step. In the step, we apply the CHILD-AND rule:

$$\begin{array}{c}
\{\langle \text{dv.gen-id}^I, \perp \rangle\}; \Sigma^0 \xrightarrow{a} \{\langle \text{dv.gen-id}^I, \perp \rangle\}; \Sigma^0 \\
\{\langle \text{post.create}^I, \perp \rangle\}; \Sigma^0 \xrightarrow{a} \{\langle \text{post.create}^I, \perp \rangle\}; \Sigma_{post}^1 \\
\{\langle \text{event.create}^I, \perp \rangle\}; \Sigma^0 \xrightarrow{a} \{\langle \text{event.create}^I, \perp \rangle\}; \Sigma_{event}^1 \\
\hline
\text{CHILD-AND} \\
\{\langle \prec \emptyset, C, B, \text{submit}_\wedge \succ, \perp \rangle\}; \Sigma^0 \longrightarrow \\
\{\langle \prec \emptyset, C, B, \text{submit}_\wedge \succ, \perp \rangle\}; \Sigma^1
\end{array}$$

where

$$\Sigma_{post}^1 = \left\{ \begin{array}{l} \text{dv} \mapsto \emptyset \\ \text{post} \mapsto \{\text{posts} \mapsto \\ \quad \{(\text{id}: 8afc, \\ \quad \quad \text{content: "New exhibit at the MIT Museum!"})\}\} \\ \text{event} \mapsto \{\text{events} \mapsto \emptyset\} \end{array} \right\} \quad (6.12)$$

$$\Sigma_{event}^1 = \left\{ \begin{array}{l} \text{dv} \mapsto \emptyset \\ \text{post} \mapsto \{\text{posts} \mapsto \emptyset\} \\ \text{event} \mapsto \{\text{events} \mapsto \\ \quad \{(\text{id}: 8afc, \text{start}: 03/03/2020, \text{end}: 03/05/2020)\}\} \end{array} \right\} \quad (6.13)$$

and  $\Sigma^1$  is the result of  $\Sigma^0 \otimes \Sigma_{post}^1 \otimes \Sigma_{event}^1$ :

$$\Sigma^1 = \left\{ \begin{array}{l} \text{dv} \mapsto \emptyset \\ \text{post} \mapsto \{\text{posts} \mapsto \{ \\ \quad (\text{id}: 8afc, \text{content: "New exhibit at the MIT Museum!"})\}\} \\ \text{event} \mapsto \{\text{events} \mapsto \\ \quad \{(\text{id}: 8afc, \text{start}: 03/03/2020, \text{end}: 03/05/2020)\}\} \end{array} \right\} \quad (6.14)$$

$\langle application \rangle ::= \langle compDef \rangle^*$	<i>define application</i>
$\langle compDef \rangle ::= [ \text{'tx'} ] component\text{-}name \text{' := ' } \langle component \rangle$	<i>define component</i>
$\langle component \rangle ::= property\text{-}name^* \langle binding \rangle^* \text{' [ ' } \langle child \rangle^* \text{' ] '}$	<i>component value</i>
$\langle child \rangle ::= [ concept\text{-}name \text{' . ' } ] component\text{-}name \langle binding \rangle^*$	<i>include component</i>
$\langle binding \rangle ::= property\text{-}name \text{' = ' } expr$	<i>property binding</i>

Figure 6-4: Core Déjà Vu syntax in Backus-Naur form

The two concept components, `post.create` and `event.create`, are synchronized to execute an action at the same time because they are included in a transaction component. Each concept component modifies their own concept server state. The different application server states  $\Sigma_{post}^1$  and  $\Sigma_{event}^1$ , one with the `post` state updated and the other one with the `event` state updated, are then merged together to form the new application server state.

## 6.5 Core Syntax and Translation

A grammar in Backus-Naur form that describes a core syntax of Déjà Vu is shown in Fig. 6-4. Non-terminal symbols are enclosed between  $\langle \rangle$  and symbols between quotes are terminals. The strings *component-name*, *concept-name* and *property-name* represent names in  $\mathcal{N}$ ,  $\mathcal{T}$ , and  $\mathcal{P}$  respectively. The string *expr* represents an expression in the syntax  $\mathcal{E}$ . In addition to the standard Backus-Naur form operators, we use  $x^*$  to denote zero or more occurrences of  $x$ , and  $[ x ]$  for an optional  $x$ . We use  $\mathcal{P}(A)$  to denote the powerset of set  $A$ . The core Déjà Vu syntax is the same we used as pseudocode for the examples.

The semantic functions that translate the syntactic constructs of Fig. 6-4 to a set of application component instance configurations are shown in Fig. 6-5. The semantic functions make use of the catalog function  $\Theta_{CC}$  to retrieve the initial concept component instance configuration for a child concept component. Given a Déjà Vu application  $a$ , we run  $A_{AC}[[a]]\emptyset$  to obtain the set  $AC_0$ .

$$\begin{aligned}
A_{\mathbb{A}\mathbb{C}} &: \langle \text{application} \rangle \rightarrow \text{Store} \rightarrow \wp(\mathbb{A}\mathbb{C}) \\
A_{\mathbb{A}\mathbb{C}} \llbracket t_1 n_1 := c_1 \dots t_m n_m := c_m \rrbracket \Gamma &\equiv \\
&\text{let } G(x) = G(x-1) \cup \{n_x \mapsto D_{\mathbb{A}\mathbb{C}} \llbracket t_x n_x := c_x \rrbracket G(x-1)\}, G(0) = \Gamma \text{ in} \\
&\text{img } G(m)
\end{aligned}$$

$$\begin{aligned}
D_{\mathbb{A}\mathbb{C}} &: \langle \text{compDef} \rangle \rightarrow \text{Store} \rightarrow \mathbb{A}\mathbb{C} \\
D_{\mathbb{A}\mathbb{C}} \llbracket \mathbf{tx} \ n := c \rrbracket \Gamma &\equiv \\
&\text{let } \langle \{p_1, \dots, p_n\}, \mathcal{C}, B \rangle = T_{\mathbb{A}\mathbb{C}} \llbracket c \rrbracket \Gamma \otimes \{ \mathbf{this} \mapsto n \} \text{ in} \\
&\prec \{p_1 \mapsto \perp, \dots, p_n \mapsto \perp\}, \mathcal{C}, B, n_{\wedge} \succ \\
D_{\mathbb{A}\mathbb{C}} \llbracket n := c \rrbracket \Gamma &\equiv \\
&\text{let } \langle \{p_1, \dots, p_n\}, \mathcal{C}, B \rangle = T_{\mathbb{A}\mathbb{C}} \llbracket c \rrbracket \Gamma \otimes \{ \mathbf{this} \mapsto n \} \text{ in} \\
&\prec \{p_1 \mapsto \perp, \dots, p_n \mapsto \perp\}, \mathcal{C}, B, n_{\vee} \succ
\end{aligned}$$

$$\begin{aligned}
T_{\mathbb{A}\mathbb{C}} &: \langle \text{component} \rangle \rightarrow \text{Store} \rightarrow \wp(\mathbb{P}) \times \wp(\mathbb{C}) \times \wp(\mathbb{B}) \\
T_{\mathbb{A}\mathbb{C}} \llbracket i_1 \dots i_n \ p_1 = e_1, \dots, p_m = e_m \ [d_1 \dots d_k] \rrbracket \Gamma &\equiv \\
&\text{let } \langle c_1, B_1 \rangle = C_{\mathbb{A}\mathbb{C}} \llbracket d_1 \rrbracket \Gamma, \dots, \langle c_k, B_k \rangle = C_{\mathbb{A}\mathbb{C}} \llbracket d_k \rrbracket \Gamma \text{ in} \\
&\langle \{i_1, \dots, i_n, \Gamma[\mathbf{this}].p_1, \dots, \Gamma[\mathbf{this}].p_m\}, \\
&\quad \{c_1, \dots, c_k\}, \\
&\quad \{\Gamma[\mathbf{this}].p_1 \leftarrow e_1, \dots, \Gamma[\mathbf{this}].p_m \leftarrow e_m\} \cup \cup_{i=1}^m B_i \rangle
\end{aligned}$$

$$\begin{aligned}
C_{\mathbb{A}\mathbb{C}} &: \langle \text{child} \rangle \rightarrow \text{Store} \rightarrow \mathbb{C} \times \wp(\mathbb{B}) \\
C_{\mathbb{A}\mathbb{C}} \llbracket n \ p_1 = e_1 \dots p_m = e_m \rrbracket \Gamma &\equiv \langle \Gamma[n], \{n.p_1 \leftarrow e_1, \dots, n.p_m \leftarrow e_m\} \rangle \\
C_{\mathbb{A}\mathbb{C}} \llbracket c.n \ p_1 = e_1 \dots p_m = e_m \rrbracket \Gamma &\equiv \langle \Theta_{\mathbb{C}\mathbb{C}}(c.n), \{c.n.p_1 \leftarrow e_1, \dots, c.n.p_m \leftarrow e_m\} \rangle
\end{aligned}$$

Figure 6-5: Semantic functions that translate grammar rules to a set of initial application instance configurations

$$\begin{aligned}
A_{\Sigma} &: \langle \text{application} \rangle \rightarrow \wp(\mathcal{T} \times \mathcal{S}) \\
A_{\Sigma} \llbracket t_1 n_1 := c_1 \dots t_m n_m := c_m \rrbracket &\equiv \{c \mapsto \Theta_{\mathcal{S}}(c) \mid c \in D_{\Sigma} \llbracket c_1 \rrbracket \cup \dots \cup D_{\Sigma} \llbracket c_m \rrbracket\} \\
T_{\Sigma} &: \langle \text{component} \rangle \rightarrow \wp(\mathcal{T}) \\
T_{\Sigma} \llbracket i_1 \dots i_n \ p_1 = e_1, \dots, p_m = e_m \ [d_1 \dots d_k] \rrbracket &\equiv C_{\Sigma} \llbracket d_1 \rrbracket \cup \dots \cup C_{\Sigma} \llbracket d_k \rrbracket \\
C_{\Sigma} &: \langle \text{child} \rangle \rightarrow \wp(\mathcal{T}) \\
C_{\Sigma} \llbracket n \ p_1 = e_1 \dots p_m = e_m \rrbracket &\equiv \emptyset \\
C_{\Sigma} \llbracket c.n \ p_1 = e_1 \dots p_m = e_m \rrbracket &\equiv \{c\}
\end{aligned}$$

Figure 6-6: Semantic functions that translate grammar rules to an initial application server state

The semantic functions that we use to obtain the initial server state  $\Sigma^0$  are shown in Fig. 6-6. Given a Déjà Vu application  $a$ , we run  $A_\Sigma[[a]]$  to obtain  $\Sigma^0$ . The initial application instance configuration is  $\emptyset; \Sigma^0$ . New clients can be added to the application instance through the CLIENT-ADD rule.

## 6.6 Other Considerations

Compared to our core syntax, the full language supports defining routes, setting the input values of components to navigate to, configuring concepts, including a same concept or component multiple times, and allowing concept components to store information in the local storage of the browser. Although the core language is much smaller than the full Déjà Vu language, it captures its semantic essence, and it's relatively straightforward to extend it to support these features and to translate the constructs of the full language into the core language.

## 6.7 Summary

In this chapter, we presented a formal semantics of Déjà Vu. The semantics explain the behavior of an application instance as a sequence of steps the application instance can perform. We formally defined relevant notions in Déjà Vu such as components, application instances and clients. We also presented a core syntax of Déjà Vu and showed how to translate it to the mathematical objects used in our semantics. Although the core syntax is much smaller than the full Déjà Vu language, it captures the semantic essence of Déjà Vu. In the next chapter, we explain how we implemented the semantics of Déjà Vu.

THIS PAGE INTENTIONALLY LEFT BLANK

# Chapter 7

## Platform Implementation

In this chapter, we describe the implementation of our Déjà Vu platform. Déjà Vu is built using TypeScript<sup>1</sup>, Angular<sup>2</sup> and Node.js<sup>3</sup>. The implementation consists of:

- a client-side library to synchronize components (§7.1);
- a server gateway to coordinate transactions and run security checks (§7.2);
- a compiler that transpiles a Déjà Vu application into an Angular application (§7.3); and
- a catalog of concepts (§7.4).

We also have a few Angular components that implement built-in Déjà Vu components such as `dv.gen-id` for generating identifiers, `dv.link` for creating links, and `dv.button` for creating buttons that trigger the execution of a transaction.

Fig. 7-1 shows the architecture of Déjà Vu. Each concept (A, B, or C) has a collection of client-side components, a server, and a database. The software elements that are part of our platform (DV) are the client-side library and the server gateway. The client-side library communicates with the gateway, which then communicates with the concept servers. The communication between a concept server and its database

---

<sup>1</sup><https://www.typescriptlang.org/>

<sup>2</sup><https://angular.io>

<sup>3</sup><https://nodejs.org>

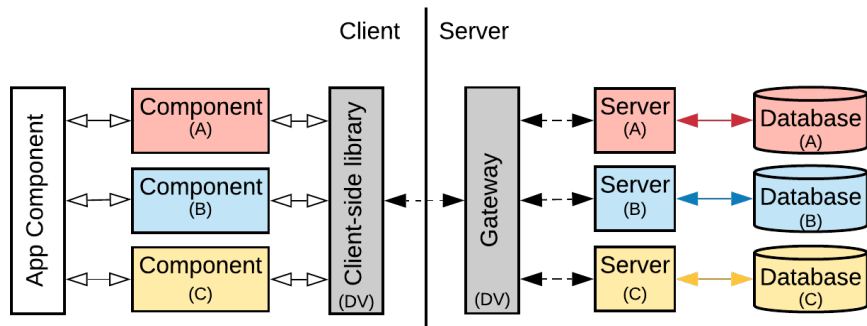


Figure 7-1: Architecture of Déjà Vu

requires no mediation. The communication between the components and the client-side library (Fig. 7-1,  $\leftrightarrow$ ) happen within the web browser. The communication between the client-side library and the gateway, and between the gateway and the concept servers (Fig. 7-1,  $\leftrightarrow$ ), happen via HTTP. A developer could, with very little code, modify the gateway and concept servers to use HTTPS instead of HTTP if desired. The gateway communicates with concept servers through designated routes all concept servers are required to implement.

In principle, there's no reason why the gateway software program and concept server programs could not be replicated in multiple physical machines for scalability or reliability. In fact, even with our current implementation, a skilled developer could make this change with a few lines of code, but our platform currently has no explicit support for configuring the number of replicas of the gateway or concept servers. The default behavior of our platform is to co-locate the gateway and concept servers on the same physical machine to reduce network latency, but it is easy for a developer to deploy the gateway and concept servers on different machines or to co-locate only some of them. Nonetheless, even when the gateway and concept servers are co-located on the same physical machine, they still communicate via HTTP.

## 7.1 Client-Side Library

The client-side library has two responsibilities: dispatch eval/exec requests to the appropriate concept components (§7.1.1) and communicate with the gateway (§7.1.2).



### 7.1.1 Event Dispatching

The client-side library allows concept components to register with the runtime system to get notified when they should eval/exec, and to request the system to trigger eval/execs of other components. The library is an event mediator [62]: a concept component doesn't subscribe to eval/exec events announced by other components directly, but it does so indirectly through the library. The library determines how to dispatch an eval/exec event depending on whether the application component is a transaction component or not. Thus, it is as if each application component has a local mediator to coordinate its own synchronization.

If the application component is a transaction component, the client-side library dispatches an eval/exec request event of a concept component to all the concept components in the application component. If the application component is a regular component, the client-side library dispatches an eval/exec request event only to the concept component that issued the eval/exec request.

### 7.1.2 Client-Server Communication

The client-side of a concept doesn't communicate with its server directly. All communication happens through the client-side library, which then communicates with the gateway over HTTP. When a component evals/execs, the component tells the runtime system which inputs were provided and can give extra information for its concept server.

If the concept component being run is not part of a transaction, the client-side library issues an HTTP request to the gateway as soon as the run request is received. If it is part of a transaction, the client-side library triggers the eval/exec of the other concept components in the application component, batches all run requests from all components that are part of the transaction, and sends only one aggregate request to the gateway.

After the gateway processes the request, concept servers receive an HTTP request with the name of the component to run, whether it's an eval or exec, its inputs, and the extra information provided by the component.

## 7.2 Gateway Server

The gateway receives, from the client-side library, the information on what component executed given as a path from the root, with what inputs, and the extra information provided by the component. At this point, the gateway runs security checks to ensure that the request is valid (§7.2.1). If the request is invalid, the gateway returns an error. If the request is valid, the gateway forwards the request or initiates a transaction if the request is a transaction request (§7.2.2).

### 7.2.1 Security

Our runtime system has no built-in notion of authentication or authorization. This functionality is implemented in concepts, which enables experts to author a variety of concepts that implement different authentication and authorization mechanisms without requiring changes to the runtime system. The server-side concept implementations are part of the trusted computing base [35] of our platform, and are assumed to have not been compromised. But the client-side code, of course, cannot be trusted. We therefore need to ensure that a client cannot violate the structure of transactions, or run the server-side action of a component that is not included in the application.

There are three properties of our implementation that allow the platform to enforce a policy: (1) when run, components report the input values that they were run with; (2) the gateway knows what components are expected to run and what the input bindings are; and (3) concept components don't communicate with their concept servers directly, but rather all communication is via the gateway.

On startup, our system provides the application source code to the gateway. From the source code, the gateway builds a component tree for each application route. For example, an excerpt of the component tree for the "/" route of *Slacker News*

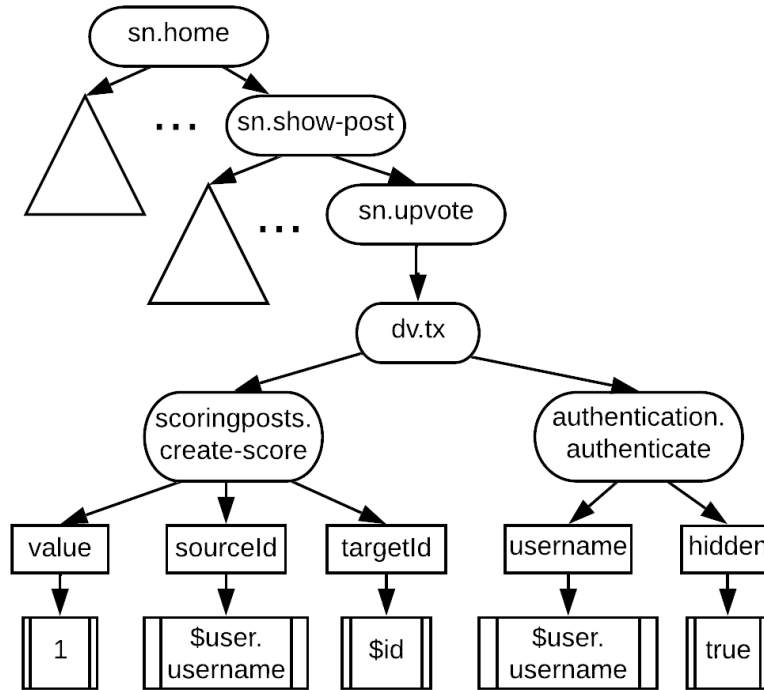


Figure 7-2: Excerpt of the component tree for route "/" of *Slacker News*

is shown in Fig. 7-2. Each component tree records the hierarchical relationships between all the components that are reachable from the component the route maps to. In *Slacker News's* configuration file (Fig. 5-6), the route "/" is mapped to the `home` component, which is why `sn.home` is the root of the component tree shown in Fig. 7-2. The component `sn.show-post` is a child of `sn.home`, `sn.upvote` is a child of `sn.show-post`, and so on. In addition to recording the hierarchical relationship between components, the component tree also records the input property bindings. Thus, for example, `sourceId` is an input of `scoringposts.create-score` and it is bound to the expression `$user.username`.

When the gateway receives a request to `eval/exec` a certain component, it performs a series of security checks. We will discuss the checks for transaction requests first and discuss non-transaction requests later. First, when the gateway receives a transaction request, the gateway verifies that the component path argument given by the client-side library is a path of some component tree of the application. For example, let's say that the gateway gets a request to execute the `upvote` transaction of *Slacker*

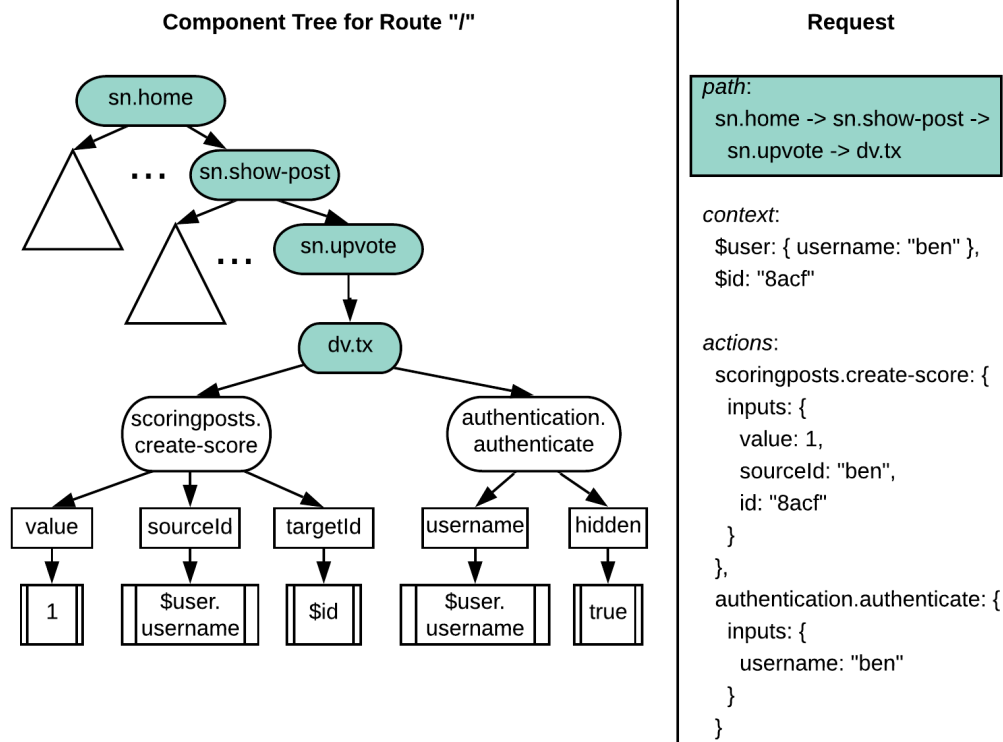


Figure 7-3: Component path check

*News*. Then, as shown in Fig. 7-3, the gateway checks that the request path `sn.home`  $\rightarrow$  `sn.show-post`  $\rightarrow$  `sn.upvote`  $\rightarrow$  `dv.tx` is a valid path. In this case, the path is a valid path because it is a path of the component tree for route `"/"`. This check prevents a malicious user from executing a server-side action that is not part of the application. For example, since `scoringposts.delete-score` is not part of *Slacker News*, there's no component tree path that includes `delete-score`, and it is not possible for a malicious user to delete a score.

The second check the gateway performs on a transaction request is to verify that the transaction request has a valid structure. A transaction request has a valid structure if, for each concept component that is a child of the `dv.tx` node in the component tree, there is a corresponding action in the request. As shown in Fig. 7-4, for the `upvote` example the gateway checks that the request includes information about `scoringposts.create-score` and `authentication.authenticate`. The transaction request must include information about `scoringposts.create-score`

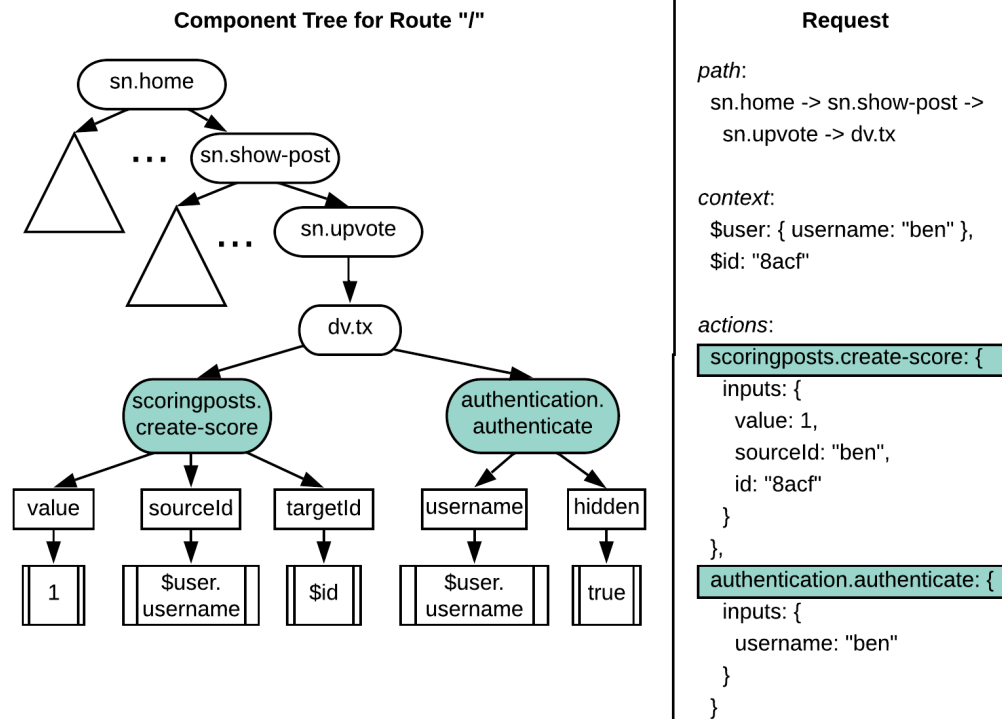


Figure 7-4: Transaction structure check

and `authentication.authenticate` because these components are the only two children of the `dv.tx` node in `sn.upvote`. This check prevents a malicious user from executing an incomplete transaction, which would allow a malicious user to, for example, remove an authentication check from a transaction to perform an operation it doesn't have the necessary permissions to perform.

Finally, the gateway checks that input values received for the server-side actions are consistent with what is specified in the component tree. It does this by evaluating each input binding under the request context and verifying that the result matches the input value reported by the component. As shown in Fig. 7-5, for the `upvote` example the gateway evaluates the input binding of `value`, `sourceId`, `targetId`, and `username` under the request context, and checks that the result matches the input values reported by `scoringposts.create-score` and `authentication.authenticate`. As a result of this check, a malicious user can't, for example, add 10 points to a post in one `upvote` because 10 is not equal to 1. Also, because the `sourceId` input

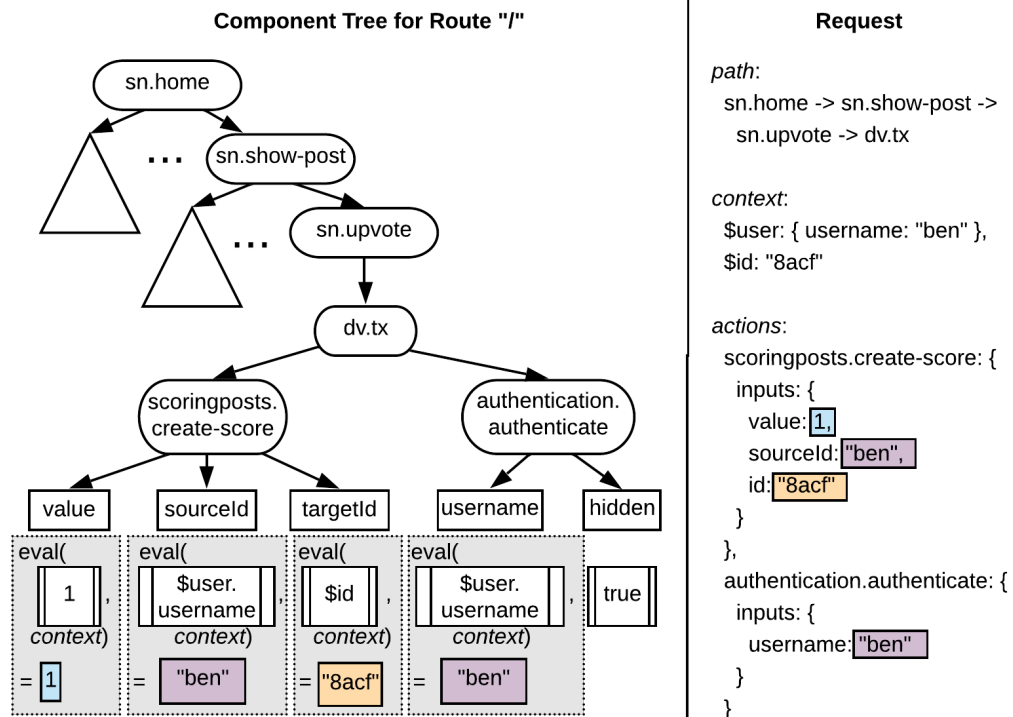


Figure 7-5: Input check

of `create-score` and the `username` input of `authenticate` are bound to the same expression `$user.username`, a malicious user can't provide a user for `authenticate` that is different from the one used as the source of the score in `create-score`. The `hidden` input is not checked server-side because the only effect of a `hidden` input is to hide a component client-side.

While evaluating code with untrusted inputs sounds dangerous, note that the code being evaluated, a template expression, is very restricted in terms of what it can do: while the syntax of template expressions is similar to that of JavaScript expressions, no function calls or JavaScript operators that produce side-effects are allowed. Also, the gateway evaluates the expression in a sandbox<sup>4</sup> that prevents the code from accessing the executing environment.

<sup>4</sup><https://www.npmjs.com/package/vm2>

For non-transaction requests, the gateway performs the same checks, with the exception of the check that the transaction has the right structure, which is not necessary because the request is not a transaction.

### 7.2.2 Transactions

If the request is valid and it is a non-transaction request, the gateway forwards the request to the corresponding concept server and forwards the response obtained from the concept server back to the client-side library.

If the request is a transaction request, the gateway acts as a transaction coordinator and runs a two-phase commit [22] with all the concept servers that are part of the transaction. The two-phase commit protocol is a popular communication protocol for ensuring that the effects of a distributed transaction are atomic, so that either all the effects of a transaction persist or none do. The two-phase commit protocol consists of two phases: a voting phase and a commit phase. In the voting phase, a transaction coordinator attempts to prepare all the participants to take the necessary steps for the participants to commit or abort the transaction. Each participant votes `ok` if they can commit, or `abort` if they detect a problem with the transaction. In the commit phase, the coordinator commits the transaction if all participants voted `ok`, or aborts if at least one participant voted to `abort`, and notifies the result to all participants. The participants then proceed to commit or abort the changes.

In Déjà Vu, we use the two-phase commit protocol to ensure all concept servers agree that a transaction is valid before any of the concepts involved in the transaction make a permanent change to their server state. If all concept servers vote `ok`, the gateway commits the transaction and forwards all the responses from the concept servers in one HTTP response to the client-side library. The client-side library demultiplexes the gateway response and forwards the individual responses to the concept components. If at least one concept server votes `abort`, the gateway sends `abort` messages to all concept servers and forwards the responses from the concept servers that voted `abort` back to the client-side library. The error responses are used client-side by concept components to show an error to the end-user. Note that the responses

from the concept servers that voted `ok` are not sent back if the transaction aborts. This is to prevent a malicious client from receiving information it is not allowed to see.

## 7.3 Compiler

Our compiler outputs an Angular application from the configuration and component files of the Déjà Vu application. For each Déjà Vu application component, it creates an Angular component. Our component language is a very thin layer atop Angular's template syntax. The data binding functionality that recomputes a template expression when any of its data dependencies changes is implemented by the Angular framework.

When a developer runs a Déjà Vu application, we run the compiler, save the output of the compiler in a hidden directory, and start the gateway and the concept servers. The gateway, in addition to processing `eval/exec` requests, serves the Angular application generated by the compiler.

## 7.4 Concept Catalog

Table 7.1 shows the current state of our catalog. To give a sense of the amount of functionality implemented in each concept, we include the number of components (#C) and the number of lines of HTML, CSS, client- and server-side TypeScript code in the concept's implementation (LoC). The lines of code count includes comments and blank lines, but no unit tests are counted.

Most of the catalog functionality was implemented to replicate the student applications of our case study. Many of the student applications are social applications and therefore our catalog includes several concepts, such as *Comment*, *Label* and *Rating*, that are commonly found in social applications [4]. This catalog is, of course, just a preliminary version. We hope expert developers will grow the catalog and contribute new concepts. What the catalog will look like as more applications are developed



Table 7.1: Concept implementations in our catalog

Concept	Purpose	# C	LoC
Authentication	Verify a user’s identity with a username and password	10	1,105
Authorization	Control access to resources	12	1,191
Chat	Exchange messages in real time	5	684
Comment	Share reactions to items	6	819
Event	Schedule events	8	1,116
Follow	Receive updates from sources	13	1,212
Geolocation	Locate points of interest	8	1,248
Group	Organize members into groups so that they can be handled in aggregate	13	1,247
Label	Label items so that they can be found later	9	1,020
Match	Connect users after they both agree	8	859
Passkey	Verify a user’s identity with a code	6	587
Property	Describe an object with properties that have values	12	2,434
Ranking	Rank items	5	574
Rating	Crowdsourcing evaluation of items	9	929
Schedule	Find a time to meet	8	1,691
Scoring	Keep track of scores	7	970
Task	Keep track of pieces of work to be done	13	1,310
Transfer	Transfer money or items between accounts	12	1,207

remains to be seen—we discuss this question and other open questions regarding the nature of our catalog later in Chapter 10. To support concept development, we have built a command-line tool for scaffolding concepts and various libraries to ease common tasks [34], such as handling transaction requests according to the two-phase commit protocol.

### 7.4.1 Authoring Concepts

Concept authors implement a server file to process gateway requests and an Angular component for each concept component. Each Angular component imports our client-side library and invokes a library method to register itself with the runtime system as soon as it loads. The same client-side library can be used by the component to trigger the eval/exec. Components define callback methods that are invoked by our system

when there's an `exec/eval` event. Within an `exec/eval` method, the component can block for inputs.

While our current implementation is tied to Angular, it might be possible to create a framework-agnostic version of Déjà Vu that would allow concept authors to use whatever client-side framework they are most familiar with. We will discuss these and other potential improvements to make it easier to author concepts later in Chapter 10.

## 7.4.2 Criteria for Creating Concepts

To avoid overlapping functionality between concepts, we only add a new concept to the catalog if there is no other concept with a similar purpose, and we only add functionality to a concept if such functionality cannot be obtained by combining the concept with other concepts. But having simpler and more orthogonal concepts can mean more work combining them. For example, *Authentication* does not include assigning first and last names to users, since this functionality can be obtained by including *Property* to an application. It would be easier for application developers, however, to have such common features included in *Authentication* as a configuration option despite the redundancy. The right balance will have to be found empirically. This trade-off between orthogonality and convenience happens in any system with software components. In general, software components tend to start small but then grow. We don't expect Déjà Vu to be any different in this sense.

A different question is whether it is desirable for the catalog to contain multiple variants of a single concept as separate concepts. For example, a variant of *Authentication* in which an email address is the primary identifier and a variant in which a social security number is used instead could be implemented as different authentication concepts. Our current approach has been to implement concept variants within a concept, and let the developer configure which variant to use through a configuration variable. For example, the transfer concept can be configured to transfer items between accounts or money. Instead of having separate *TransferItem* and *Trans-*

*ferMoney* concepts, we have only one *Transfer* concept, which can be configured to transfer items or money between accounts.

The same criteria that apply to conventional web application development apply to concept development as well, from all the user interface principles, such as [47, 42], to the software engineering principles that focus on code, such as [48, 49, 32]. Regarding the design of concepts themselves, we have previously developed a set of principles [28, 29, 50, 11] that we apply in the development of our catalog. These include the one-to-one principle that states that a concept must have exactly one purpose that motivates it; the genericity principle that states that reusing a well-known generic concept is usually preferable to inventing a new concept; and the uniformity principle that states that the behavior of a concept should be uniform so that the same actions of a concept can be applied irrespective of the context. We have designed all the concepts in our catalog to have exactly one motivating purpose, to be generic so that they can be applied in different contexts, and to be uniform so that the same concept actions can be used consistently in different contexts.

## 7.5 Summary

There are four software components in our platform implementation: a client-side library, a server gateway, a compiler, and the catalog of concepts. The client-side library synchronizes the execution of concept components and mediates client-server communication. The gateway checks that client requests correspond to what is specified in the application source code, and runs a two-phase commit protocol with the concept servers if necessary. Our compiler takes as input the JSON configuration file, the HTML files of application components, and the CSS/SASS style sheet files, and outputs an Angular application. The gateway, in addition to processing client requests, also servers the Angular application.

In the next chapter, we present our case study, in which we used *Déjà Vu* to replicate a series of non-trivial sample applications.

THIS PAGE INTENTIONALLY LEFT BLANK

# Chapter 8

## Case Study

To evaluate Déjà Vu, we conducted a case study in which we used our platform to replicate a collection of applications previously built by students for a web programming course. Through the case study, we sought to understand whether it is possible to build a variety of non-trivial applications using our platform, to compare the effort required to build an application using Déjà Vu instead of general-purpose tools, and to understand how the quality of applications built with Déjà Vu compare to applications built using standard general-purpose tools (§8.1).

To this end, we ran the student projects to determine their behaviors (§8.2) and replicated the student applications (§8.3) using Déjà Vu (§8.4). Then, we conducted a modularity analysis to determine how the Déjà Vu applications we built use the concept catalog (§8.5), we estimated the effort savings from using Déjà Vu to build the study applications instead of general-purpose tools (§8.6), and we compared the quality of the Déjà Vu and student implementations (§8.7).

### 8.1 Research Questions

The research questions we sought to answer were the following:

- *RQ1*. Is it possible to build a variety of non-trivial applications using Déjà Vu, without building non-generic concepts that are specific only to a given application?

Obviously an entire application could be built as a single concept, so the real question is not whether a given application can be built using Déjà Vu, but whether a range of applications can be constructed from a relatively small set of concepts.

- *RQ2*. How does the effort required to build a Déjà Vu application compare to using standard general-purpose tools?

Even if we can achieve a good level of concept reuse, the effort savings from concept reuse might not be enough if implementing and combining concepts is much harder than re-implementing concepts from scratch using general-purpose tools.

- *RQ3*. How does the quality of Déjà Vu applications compare to those applications built with standard general-purpose tools?

A developer can always, with enough time, fix every usability issue in an application irrespective of whether it was built using Déjà Vu or built using general-purpose tools. The question is really whether building applications with Déjà Vu tends to produce more usable applications. Are certain kinds of usability errors harder to make using Déjà Vu? What about other quality attributes such as security and performance?

## 8.2 Method

With the students' permission, we obtained access to their code repositories so we could run their applications and explore their behaviors. If we were unable to run the student application locally, we looked at the team design documents and the application code to determine the behavior of the application. For each project, we developed any missing concepts in order to replicate the behavior of the original student application. We replicated only the core functionality, omitting behavioral details that are not essential to the working of the application. We didn't use any of

Table 8.1: Student projects we replicated in Déjà Vu

Application	Purpose	LoC
Accord	Support musical bands in the selection of setlists	8,671 <sup>†</sup>
ChoreStar	Make it easy for parents to assign chores to children	3,183 <sup>*</sup>
EasyPick	Recommend classes to college students	3,161 <sup>*</sup>
GroceryShip	Facilitate peer grocery delivery between students	4,996 <sup>*</sup>
Lingua	Develop language skills by chatting with native speakers	4,639 <sup>†</sup>
Listify	Crowdsource opinion-based rankings of anything	5,876 <sup>†</sup>
LiveScorecard	Provide a live leaderboard for climbing competitions	8,742 <sup>†</sup>
MapCampus	Allow students to plan events on campus	3,807 <sup>†</sup>
Phoenix	Help people discuss mental health and make friends	7,062 <sup>*</sup>
Potluck	Help people plan parties where guests bring supplies	4,344 <sup>†</sup>
Rendezvous	Plan public events on campus	4,498 <sup>‡</sup>
SweetSpots	Mark spots on a map and review spots added by others	3,898 <sup>†</sup>

The symbols next to the code count indicate the front-end library used: <sup>†</sup>React v15, <sup>\*</sup>Handlebars v4, <sup>‡</sup>Jade v1

the code written by students other than some HTML to provide page content such as titles and some CSS to style the appearance of the application.

### 8.3 Study Subjects

The student projects are from the Fall ‘16 offering of the web programming course. The 12 applications we replicated were selected independently by the teaching assistants of the class as the best projects out of about 30 projects. The project selection happened before we contacted students about using their projects to evaluate Déjà Vu, and the teaching assistants of the course have no relation to our research project.

The student projects were mostly 4-person projects, done for 5 weeks, with each student taking 10-20 hours per week. Thus, each project represents 200-400 person-hours of work. The names of the student projects we replicated, together with their purposes, are shown in Table 8.1. We also include the number of lines of HTML, CSS, client- and server-side JavaScript code for the student implementations.<sup>1</sup> For

<sup>1</sup>The count includes comments and blank lines. Unit tests are not counted.

building the user interface, 7 student projects used React<sup>2</sup>, 4 used Handlebars<sup>3</sup> and 1 used Jade<sup>4</sup>. All projects implemented their server-side API in the REST architectural style [16].

Students were allowed to use as many libraries, frameworks, and external web API services as they wanted to. The requirements for the project were that it should be a web application with client- and server-side code and a database, that solves a plausible problem, and that has more than just basic CRUD functionality. The number of lines of code of the project was not part of the evaluation criteria and students had no incentive to produce more code than the minimal amount of code necessary to implement their project proposal.

### 8.3.1 Project Descriptions

A description of the 12 student projects is included below:

- *Accord*. In *Accord*, users can create a group and invite other users to join. Within a group, users can create new setlists, propose songs, and rate and comment on proposed songs. Only the group creator can create new setlists, but any group member can propose, rate and comment on songs. Only the members of a group have access to the group’s setlists and songs.
- *ChoreStar*. In *ChoreStar*, there are two kinds of users: parents and children. Parents create accounts for their children, and can create rewards and chores for their children. Both rewards and chores have a monetary value: when the child marks a chore as completed and the parent approves the completion of the chore, the child earns the amount of money the chore is worth and can use that money to buy rewards. The money is not real money: the currency is called “stars” and it is only used within the application to track balances.
- *EasyPick*. In *EasyPick*, users can review a course they took along various dimensions, such as grading fairness and course load. Users can also search and

---

<sup>2</sup><https://reactjs.org/>

<sup>3</sup><https://handlebarsjs.com/>

<sup>4</sup><http://jade-lang.com/>



filter courses based on ratings. The application can recommend courses to a user based on their reviews.

- *GroceryShip*. In *GroceryShip*, users can request items, such as groceries, to be delivered to their dorm. Users input an expected price for the items and specify the tip they will be paying on delivery. Other users can claim delivery requests and buy and deliver the items. Upon delivery, the user doing the delivery is paid for the items, plus the tip.
- *Lingua*. In *Lingua*, users can find pen pals that are learning a new language and chat. In a chat room, users can select a word or phrase the other user wrote and suggest a correction. Users can also rate their conversation, which determines the rating of the other user in the conversation.
- *Listify*. In *Listify*, users can submit lists of items and rank them in order of preference. For example, a user could create a new list of basketball players and submit five famous players. Users can rank the players and see what the consensus ranking is. Users can also upvote/downvote lists and close the voting on lists so that no new rankings are submitted.
- *LiveScorecard*. In *LiveScorecard*, users can create climbing competitions and add other users to the competition as climbers. The competition creator can add new climbs to the competition and assign the climbs a score. Climbers can log in to the system, record the number of times they fell while completing a climb and mark the climb as completed. The competition score of a climber is determined by the points of the completed climbs minus the number of falls. To log in to a competition, a climber uses a code that is unique to them and that is given to them by the creator of the competition. The climb competition is also identified by a unique code. Spectators can log in to the application with the climb competition code and look at the competition scorecard.
- *MapCampus*. In *MapCampus*, users can create groups and create events for a group. Only members of the group can see the events of that group. Group

administrators can invite other users to a group. Users can also choose to leave a group if they want to.

- *Phoenix*. In *Phoenix*, users can find other users to meet and chat about various topics. Users create a profile by specifying their time availability and a set of topics they are interested in discussing. The application recommends users with overlapping topic interests. A user can express interest to chat with another user. If both users express interest in each other they are matched and can communicate via email to find a time to meet.
- *Potluck*. In *Potluck*, users can create potluck events and invite other users to the event. Guests to the event can add new supply requests and claim supplies. Each supply has a description and quantity. For example, a supply request can be 2 bottles of diet coke. A user can choose the quantity of a supply to claim. For example, a user could claim to bring only 1 bottle of diet coke and the other bottle will appear as unclaimed.
- *Rendezvous*. In *Rendezvous*, users can create public events with a location, date and time, and a set of tags. Users can view all events on a map, and mark which ones they plan to attend. Users can also comment on events.
- *SweetSpots*. In *SweetSpots*, users can create new campus spots, such as a library or a park. They can rate the spot and adds tags. Users can see all spots on a map, add reviews to a spot, upvote/downvote another user's review of a spot and mark a spot as favorite. Users have a reputation score, which is determined by the number of reviews the users submitted and the number of times the review was upvoted/downvoted by users.

## 8.4 Study Replicas

We used Déjà Vu to replicate the core functionality of all 12 student applications. The code for the Déjà Vu implementations of the student projects is available online.<sup>5</sup> We

---

<sup>5</sup> <https://deja-vu-platform.com>

did not replicate behavior that was clearly anomalous and we did not replicate MIT-specific functionality that would make our sample applications unusable to anyone outside of the MIT community.

For example, in the login page of *ChoreStar*, the student implementation has a button with a label “Click here if not redirected” for the end-user to click on it if the automatic redirect that should happen after a successful log-in doesn’t work. This is probably a bug in the student implementation that the students couldn’t fix by the project due date. It would be silly for us to replicate the bug and add the button. Instead, our Déjà Vu implementation of *ChoreStar* correctly redirects the user after a successful login, and doesn’t include the redirect button. Behavior that we didn’t replicate and that is less clear whether it is anomalous or not is discussed in detail later in §8.7.

Five student projects use the MIT API for authentication and restrict the application to MIT-students only. The student projects are *EasyPick*, *GroceryShip*, *Rendezvous*, *MapCampus*, and *SweetSpots*. We could have replicated this functionality, but we chose not to so as to make our Déjà Vu implementations usable by everyone and not just usable by people affiliated with MIT.

When discussing our study results we will mention and account for the functionality differences between the student implementations and the Déjà Vu implementations when applicable.

## 8.5 Modularity Analysis

In this section, we analyze how the Déjà Vu applications we built to replicate student projects use the catalog. The applications in the suite, together with the number of times they use a concept from the catalog, are shown in Table 8.2.

The median number of concept types used per applications is  $Q_2=6$  ( $Q_1=5.75$ ,  $Q_3=8$ ). The median number of concept instances used per applications is  $Q_2=9$  ( $Q_1=7.75$ ,  $Q_3=10$ ). Most applications use roughly the same number of concept instances ( $\sigma=2.3$ ). This is probably because all the student projects we replicated

Table 8.2: Concept usage in sample applications

Concept/application	Accord	Chorestar	EasyPick	GroceryShip	Lingua	Listify	LiveScorecard	MapCampus	Phoenix	Potluck	Rendezvous	SweetSpots	#Applications	#Instances
Authentication	1	2	1	1	1	1	1	1	1	1	1	1	12	13
Authorization	1	1	1	1	1	1	1	2	1	1	1	1	12	13
Chat	0	0	0	0	1	0	0	0	0	0	0	0	1	1
Comment	1	0	1	0	0	0	0	0	1	0	1	1	5	5
Event	0	0	0	0	0	0	1	1	0	1	1	0	4	4
Follow	0	0	0	0	0	0	0	0	0	0	0	1	1	1
Geolocation	0	0	0	0	0	0	0	1	1	0	1	1	4	4
Group	1	0	0	0	2	1	3	1	0	1	1	0	7	10
Label	0	0	0	0	0	0	0	0	1	0	1	1	3	3
Match	0	0	0	0	0	0	0	0	1	0	0	0	1	1
Passkey	0	0	0	0	0	0	2	0	0	0	0	0	1	2
Property	5	3	3	3	2	2	4	3	2	2	2	1	12	32
Ranking	0	0	0	0	0	1	0	0	0	0	0	0	1	1
Rating	1	0	4	1	1	0	0	0	0	0	0	1	5	8
Schedule	0	0	0	0	0	0	0	0	1	0	0	0	1	1
Scoring	0	0	0	0	0	1	2	0	0	0	0	2	3	5
Task	0	1	0	0	0	0	1	0	0	0	0	0	2	2
Transfer	0	1	0	0	0	0	0	0	0	1	0	0	2	2
# Concept Types	6	5	5	4	6	6	8	6	8	6	8	9		
# Concept Instances	10	8	10	6	8	7	15	9	9	7	9	10		

took a similar amount of person-hours of work to develop, and are therefore roughly equivalent in terms of complexity.

The median number of times a concept is used across projects is  $Q_2=3$  ( $Q_1=1$ ,  $Q_3=5$ ). The median number of times a concept is instantiated is  $Q_2=3.5$  ( $Q_1=1.25$ ,  $Q_3=7.25$ ). All of the applications require identification of users and preventing them from accessing content they don't own. Also, it is very common for applications to need to store domain-specific fields. For example, a "description" for parties in *Potluck*. Thus, *Authentication*, *Authorization*, and *Property* are the most used concepts. The *Property* concept is the concept with the most number of instances. This is because applications need an instance of *Property* for each kind of entity, and a given application could have multiple entities. For example, in *Accord*, there are bands, song suggestions, setlists, media links, and user profiles.

*Chat*, *Follow*, *Match*, *Ranking*, and *Schedule* are only used once. We do not think it is because these concepts are too application-specific. For *Chat*, we think it might be because it is challenging to implement, so only one of the winning student teams risked doing so. With enough time, other projects might have ended up incorporating such functionality. For example, *Rendezvous* might have created a group chat for each campus event so that guests could talk.

*Follow* implements functionality that is very common in social media applications: subscribing to a source of updates. For example, Twitter lets you follow other accounts, and tweets from accounts you follow appear in your feed. *Ranking* lets users rank items and show the aggregate consensus ranking of items. While this looks like a rather specific concept, note that many applications for managing human resources usually include functionality like this, so that managers and stakeholders can stack rank employees to determine promotions.

*Match* and *Schedule* are only used in *Phoenix*. *Phoenix*'s functionality revolves around matching users after they both expressed interest in each other and giving a way for users that match to find a time to meet in person. While no other sample application uses *Match* or *Schedule*, many applications, such as dating sites, have

matching functionality. And many productivity applications provide functionality for scheduling meetings.

## 8.6 Effort Savings

To estimate and compare the difference in effort required to build the study applications with Déjà Vu compared to using standard general-purpose tools, we developed a metric. The metric estimates, using lines of code, the effort savings from using Déjà Vu to build an application instead of using general-purpose tools.

### 8.6.1 Metric Considerations

#### Concept Reuse

For estimating the cost of a Déjà Vu implementation, we include the cost of developing the concepts used in that application. Not doing so would be unfair to the student implementations: while we regard the concept catalog as part of our platform, someone has to implement the concepts in the first place. On the other hand, a key benefit of Déjà Vu is concept reuse. Once a concept is implemented, other applications can use it. If two applications reuse the same concept, it would be unfair to add the entire cost of developing the concept to each application. Thus, instead of looking at each application in isolation, our metric considers the cohort of applications an application is part of, and shares the cost of developing a concept equally among all other applications in the cohort that reuse the same concept. For the purpose of this case study, the cohort of applications is the 12 applications that are the subjects of the study, and we compute and discuss the effort savings for each one of the 12 applications.

#### Library Reuse

The purpose of the effort-savings estimate is to quantify the effort savings you can typically obtain from using Déjà Vu to develop an application instead of using general-

purpose tools. In general-purpose tools, effort savings come from reusing libraries and frameworks. For our effort-saving estimates to be accurate, the student implementations must have, to some extent, exhausted all the effort-saving opportunities available to them from library and framework reuse.

To investigate the extent to which student projects used libraries we: (1) counted and analyzed the libraries and frameworks used in the student projects and, (2) for each concept used by the Déjà Vu implementations, we searched the NPM package registry for libraries and the web for web APIs to see if there are any libraries or web APIs that the students could have used to implement concept functionality but neglected to do so.

To obtain a list of the libraries and frameworks used in the student projects we did the following. First, for each student implementation, we extracted the project dependencies from the `package.json` file of the project. Then, for each dependency, we read the package description in the registry to determine if the dependency was a dependency that is only needed for local development and testing. If the dependency was a development-only dependency, we didn't include the dependency in the list of libraries and frameworks used.

Table 8.3: Libraries and frameworks used in the student implementations

Package Name	Description	# Projects	Weekly Downloads
<code>body-parser</code>	Parse request bodies	12	11,180,379
<code>express</code>	Server-side web framework	12	10,755,586
<code>bcrypt</code> <sup>6</sup>	Hash passwords	12	913,332
<code>express-session</code>	Express session middleware	12	699,351
<code>mongoose</code>	MongoDB object modeling	12	692,150
<code>cookie-parser</code>	Cookie header parser	11	1,299,698
<code>request</code>	HTTP request client	10	15,644,167
<code>moment</code>	Date utilities	8	10,098,784
<code>react</code>	Front-end library	7	6,237,933
<code>request-promise-native</code>	HTTP request client	7	6,235,363
<code>react-dom</code>	DOM library for React	7	5,540,051

Continued on next page

<sup>6</sup>Project count and weekly downloads also include `bcrypt-nodejs` and `bcryptjs`

Table 8.3 – continued from previous page

Package Name	Description	# Projects	Weekly Downloads
<code>react-router</code>	Routing library	7	2,637,139
<code>nodemailer</code>	Send emails	7	928,400
<code>uuid</code> <sup>7</sup>	Generate UUIDs	6	22,599,384
<code>lodash</code>	JavaScript utility library	5	25,413,041
<code>morgan</code>	Request logger middleware	5	2,023,601
<code>csrf</code>	CSRF token middleware	5	276,064
<code>handlebars</code>	Template language	4	8,413,668
<code>dateformat</code>	Date formatting	4	4,230,411
<code>async</code>	Utility library for asynchronous code	3	24,387,986
<code>express-handlebars</code>	View engine for Express	3	115,419
<code>bluebird</code>	Promise library	2	16,479,308
<code>serve-favicon</code>	Serve a favicon	2	1,609,122
<code>react-dnd</code> <sup>8</sup>	Drag and Drop	2	855,788
<code>passport</code>	Express authentication middleware	2	707,231
<code>react-datepicker</code> <sup>9</sup>	Date picker	2	510,662
<code>react-bootstrap</code>	Components library	2	487,525
<code>passport-local</code>	Username/password authentication	2	316,307
<code>hbs</code>	View engine for Express	2	76,146
<code>xoauth2</code>	XOAuth2 token generation	2	31,924
<code>email-verification</code>	Verify email sign-up using MongoDB	2	74
<code>querystring</code>	Node's querystring module for all engines	1	8,399,397
<code>q</code>	A library for promises	1	8,042,685
<code>dotenv</code>	Load environment variables from a .env file	1	7,915,846
<code>crypto-browserify</code>	Data encryption	1	7,139,963
<code>xml2js</code>	XML to JavaScript object converter	1	6,861,963
<code>compression</code>	Compression middleware	1	6,583,978
<code>underscore</code>	Functional programming helper library	1	6,284,233

Continued on next page

<sup>7</sup>Project count also includes `node-uuid`. `node-uuid` was deprecated in favour of `uuid`<sup>8</sup>Project count and weekly downloads also include `react-dnd-html5-backend`<sup>9</sup>Project count and weekly downloads also include `react-datepicker`



Table 8.3 – continued from previous page

Package Name	Description	# Projects	Weekly Downloads
amdefine	Module loading utility	1	5,739,334
ejs	Template language	1	4,644,990
redux	State container	1	3,854,631
jsonwebtoken	JSON-based access tokens	1	3,705,835
socket.io	Realtime framework server	1	2,993,577
jquery	Library for DOM operations	1	2,880,750
react-redux	React bindings for Redux	1	2,806,391
bootstrap	Front-end framework	1	2,051,794
redux-thunk	Redux thunk middleware	1	1,662,471
mongodb	MongoDB driver	1	1,495,625
js-cookie	Cookie handling	1	1,222,481
redux-logger	Redux logger	1	640,325
cron	Execute something on a schedule	1	577,647
jade	Template language	1	580,401 <sup>10</sup>
stripe	Stripe API wrapper	1	390,778
sendgrid	Email Service	1	373,154
react-copy-to-clipboard	Copy-to-clipboard component	1	316,937
immutability-helper	Mutate a copy of data without changing the original source	1	315,441
leaflet	Map component	1	276,189
react-autosuggest	Auto-suggest component	1	263,390
nodemailer-smtp-transport	SMTP transport for Node-mailer	1	225,224
nodemailer-wellknown	SMTP service configurations	1	222,480
express-validator	Express validator middleware	1	150,913
react-cookie	Universal cookies for React	1	136,081
connect-flash	Session storage	1	124,488
react-dropdown	Dropdown component	1	75,939
react-leaflet	Map component	1	66,758
react-bootstrap-typeahead	Typeahead component	1	59,499
react-autocomplete	Autocomplete component	1	56,257
redux-promise	Redux promise middleware	1	52,897

Continued on next page

<sup>10</sup>The jade package was renamed to pug. The weekly downloads number shown is for pug.

Table 8.3 – continued from previous page

Package Name	Description	# Projects	Weekly Downloads
<code>fixed-data-table</code>	Table component	1	21,927
<code>zipcodes</code>	Zipcode database library	1	8,960
<code>react-stars</code>	Star rating component	1	6,661
<code>mongoose-deep-populate</code>	Mongoose plugin to enable deep population of nested models	1	5,489
<code>react-modal-dialog</code>	Launch modal dialogs	1	1,160
<code>bootstrap-star-rating</code>	Star rating component	1	1,048
<code>mongoose-integer</code>	Validate integer values within a Mongoose Schema	1	504
<code>react-router-form</code>	Forms for React	1	259
<code>mongoose-q</code>	Mongoose with promises	1	208
<code>likely</code>	Collaborative filtering and recommendation engine	1	12
<code>http</code>	HTTP client	1	deprecated

Table 8.3 shows each distinct software library and framework used in the student implementations, together with a short description and the number of student projects using the library or framework. To give a sense of how popular each package is, we include the number of weekly downloads for each package as reported on the NPM package registry<sup>11</sup> as of January 27, 2020.

In total, the 12 student projects used 84 distinct libraries or frameworks. The total number of times any library or framework was included is 227 and the average number of inclusions per project is 19. Our count is a conservative estimate for the number of libraries and frameworks used by students because there could be dependencies that are not listed in the `package.json` file. For example, some projects include the Google Maps widget by adding some JavaScript code within an HTML file.

As shown in Table 8.3, students used popular front-end libraries and frameworks like React, Jade and Handlebars; popular server-side web frameworks and server-

<sup>11</sup><https://www.npmjs.com/>

Table 8.4: Libraries providing some concept functionality

Concept	Client-Side Functionality	Server-Side Functionality
Authentication	<code>react-loginform</code>	<code>passport</code>
Authorization	<code>react-ability</code>	<code>node-authorization</code>
Chat	<code>react-chat-elements</code>	
Comment	<code>react-commentbox</code>	
Event	<code>js-year-calendar</code>	
Follow		
Geolocation	<code>google-map-react</code>	
Group		
Label		
Match		
Passkey		
Property	<code>angular-schema-form</code>	
Ranking		
Rating	<code>react-stars</code>	
Schedule	<code>angular-calendar</code>	
Scoring		
Task		
Transfer		

side utility libraries like Express; numerous client-side components for implementing functionality such as ratings, forms, modals, tables and drag-and-drop functionality; several utility libraries for handling dates and sessions; and several libraries for data management and authentication.

Could students have used more libraries? To answer this question, we looked at each concept in our catalog and searched the NPM package registry for libraries that provide similar functionality to the client- or server-side functionality provided by the concept.

Table 8.4 shows the results of our search for existing concept functionality. Each table cell contains the name of the library we found that provides some client- or server-side concept functionality. If we found more than one library for a concept functionality, we only show the one that appears to be the most complete and popular. An empty cell indicates that we were unable to find a library for that concept functionality.

The results suggest that students wouldn't have been able to do much better in terms of library reuse, because for many concepts, such as *Task* and *Transfer*, there are no libraries available that implement the concept functionality. It is only when a concept has a widget with rich behavior, such as a calendar widget, that there exists a client-side library that implements such functionality. And it is only when the concept has complex server-side functionality, such as authenticating users or implementing a role-based authorization mechanism, that there exists a server-side library that can help implement such functionality. Also, for the cases in which a complex widget or complex server-side behavior is required, many student projects already use the same library we found or an equivalent one. For example, as shown in Table 8.3, for implementing client-side geolocation functionality student projects use `leaflet` and `react-leaflet`, for client-side rating functionality they use `bootstrap-star-rating` and `react-stars`, and for server-side authentication functionality they use `passport` and `passport-local`.

Why is it that for 9 of the 18 concepts there appears to be no libraries that implement the client- or server-side functionality of the concept? We think it is because there is not much value in having a library for only the client- or server-side functionality of these concepts. For example, the *Task* client-side behavior consists of a set of components for creating and showing tasks with a due date, assigner and assignee. These components don't add much beyond what could be quickly implemented by writing custom code using a component library like Angular Material. The value of a concept, as we've seen in §4.4, comes from the fact that it is a full-stack abstraction and that as a result it encapsulates, among other things, client-server integration code such as subscribing to events and writing event handlers, aggregating client-side data to send a request, handling client-server communication, and processing the component requests server-side. If a concept functionality is broken down into separate client- and server-side libraries, then the concept has lost most of its value, and the libraries are typically not be very valuable on their own.

Regarding web APIs, we only found full-stack APIs that implement functionality similar to *Authentication* and *Authorization* (Auth0<sup>12</sup>), *Chat* (TalkJs<sup>13</sup>) and *Comment* (Disqus<sup>14</sup>). Perhaps student implementations that include such functionality could have saved some lines of code by switching to a web API, but note that for *Authentication* and *Authorization* at least, there are several server-side libraries that the student projects are already using, and the switch to a web API might therefore provide only marginal gains—especially because the authentication and authorization functionality in the student applications is a simple username/password authentication system that is well-supported by popular libraries.

## 8.6.2 Effort Savings Metric

### Metric Definition

The metric divides the size of the Déjà Vu implementation of an application by the size of the student implementation. The size of the Déjà Vu implementation is the size of the Déjà Vu code, plus the size of the concept implementations used in the application. Since concepts are reused in multiple applications, we divide the size of a concept implementation by the number of times the concept is used by different applications in the cohort. To account for differences in languages, frameworks and functionality between the student and Déjà Vu implementations we introduce two adjustment factors. The metric is defined as follows:

$$ES(a, \mathcal{U}) = \frac{\text{LoC}(a_{dv}) + \sum_{c \in a \cdot \mathcal{U}} \frac{f_{tech} \times \text{LoC}(c)}{|\mathcal{U} \cdot c|}}{f_{\Delta} \times \text{LoC}(a_s)}$$

The parameter  $a$  represents the application being considered,  $a_{dv}$  is the Déjà Vu implementation of application  $a$ , and  $a_s$  is the student implementation of application  $a$ . The parameter  $\mathcal{U}$  is a relation that includes an  $(a, c)$  pair if application  $a$  uses concept type  $c$ .  $|S|$  denotes the cardinality of set  $S$ , and the dot a relational join.

---

<sup>12</sup><https://auth0.com/>

<sup>13</sup><https://auth0.com/>

<sup>14</sup><https://disqus.com/>

Therefore,  $a.\mathcal{U}$  is the set of distinct concept types used in application  $a$  and  $|\mathcal{U}.c|$  is the number of times a concept type  $c$  is used by applications in  $\mathcal{U}$ . The function  $\text{LoC}(p)$  computes the lines of code in program  $p$ , without counting unit tests. The count includes comments and blank lines.

The metric includes two adjustment factors,  $f_{tech}$  and  $f_{\Delta}$ , whose values are given later. The  $f_{tech}$  factor accounts for differences between the languages and frameworks used in our concepts compared to those used in the student implementations. The  $f_{\Delta}$  adjustment factor accounts for differences in functionality between the Déjà Vu and student implementations. In practice, each application would have its own set of factor values, but we simplify and use the same factor values for all applications since we don't expect the minor variations between the application-specific factor values to make a significant difference in the estimate.

## **Metric Interpretation**

In this metric, any number less than 1 indicates a positive outcome of using Déjà Vu to build the application, instead of using general-purpose tools. Since the metric shares the cost of implementing a concept with all the applications that use the same concept, there are some cases in which the real effort of using Déjà Vu as perceived by a developer could differ from the metric estimate. For example, if none of the concepts required by an application are readily available in the catalog, the developer would have to pay the cost of developing them all from scratch. Even if the developer expects those concepts to be reused by multiple applications, the developer is still doing all of the implementation work. Moreover, if the developer reusing the concepts is different from the original developer that developed them, then the original developer wouldn't be the one experiencing the subsequent effort savings. On the other hand, a developer might be able to find all the functionality they need in the catalog, and build a very complex application without implementing any concept.

### 8.6.3 Adjustment Factor Values

The values of the adjustment factors were chosen based on our observations of the student implementations and our Déjà Vu implementations, and are further explained below.

**Technology Differences.** The  $f_{tech}$  adjustment factor accounts for the technology differences between the Déjà Vu implementations and the student implementations. Our concept implementations use TypeScript, Angular, and GraphQL, while the student implementations use JavaScript, React, Handlebars or Jade, and REST. Code written in TypeScript, which is a syntactical superset of JavaScript that adds optional static typing, tends to be longer than the equivalent JavaScript code because of the extra type annotations. Angular is generally more verbose than React, Handlebars or Jade, which were the libraries used by students. Using GraphQL instead of REST typically leads to longer server-side code. This is because GraphQL requires the developer to define and implement a resolver function for each field on each type, while REST requires a developer to define a function for each endpoint and there are usually less REST endpoints than resolver functions. In GraphQL, a developer has to also define a GraphQL schema, but we don't adjust for the schema definition because our lines of code measure does not include the schema file. By observation of the student code and the equivalent concept code, we estimate that, on average, all these differences in technology result in concept code being 10% longer than the equivalent student implementation code. Hence, we normalize the concept code by multiplying it by 90% and use  $f_{tech} = 0.9$ .

**Functionality Differences.** The  $f_{\Delta}$  factor accounts for the differences in functionality between the student implementations and the Déjà Vu implementations. By interacting with the student and Déjà Vu implementations and observing the differences in functionality, we estimate that, on average, the student implementation has 10% more end-user functionality than the corresponding Déjà Vu implementation. This extra functionality in the student implementations corresponds to the non-core

Table 8.5: Déjà Vu effort savings

Application	LoC of Déjà Vu Implementation	Adjusted LoC of Used Concepts	Adjusted LoC of Student Implementation	Effort Savings Estimate
Accord	1,018	830	7,803	0.237
ChoreStar	600	1,487	2,865	0.728
EasyPick	771	669	2,845	0.506
GroceryShip	838	522	4,496	0.302
Lingua	630	1,298	4,175	0.462
Listify	1,047	1,323	5,288	0.448
LiveScorecard	1,429	2,075	7,868	0.445
MapCampus	834	947	3,426	0.520
Phoenix	1,165	3,384	6,356	0.716
Potluck	971	1,210	3,910	0.558
Rendezvous	1,502	1,401	4,048	0.717
SweetSpots	761	2,638	3,508	0.969

functionality we did not replicate. Hence, we normalize the student implementation code by multiplying it by 90% and use  $f_{\Delta} = 0.9$ . In the future, we plan to refine this number by counting the lines of code in the student implementations that correspond to the functionality we didn't replicate.

### 8.6.4 Results

The values of the effort savings metric **ES** for all applications in the case study are shown in Table 8.5. For each application, we also show the lines of code of the Déjà Vu implementation ( $\text{LoC}(a_{dv})$ ), the adjusted lines of code of the used concepts (the sum  $\Sigma$ ), and the adjusted lines of code of the student implementations ( $f_{\Delta} \times \text{LoC}(a_s)$ ). For the adjustment factors we use  $f_{tech} = f_{\Delta} = 0.9$  (§8.6.3).

### Results Discussion

The results show an estimated effort savings from using Déjà Vu in all 12 applications. The applications for which there were important effort savings have in common the fact that they: (1) instantiate the same concept multiple times, and that (2) most of the other concepts they use are widely used concepts. For example, in *Accord*



( $ES=0.237$ ), most of the effort gains probably come from the reuse of *Property*. Much of the functionality in *Accord* is CRUD functionality of different types of entities: group bands, setlists, songs, and so on. While the student implementation has specialized code to deal with the CRUD functionality of each entity type, in *Déjà Vu* the same functionality is implemented by including and configuring the same generic *Property* concept multiple times. It also helps that the other concepts used in *Accord*—*Authentication*, *Authorization*, *Comment*, *Group* and *Rating*—are concepts that are widely used by other applications so the cost of the included functionality is shared by multiple applications.

For applications in which the estimated effort savings were less, it is because there's one or more concept that the *Déjà Vu* implementations are using that are not reused as much. For *ChoreStar* for example, it is because *Task* and *Transfer* are relatively expensive concepts: they are among the concepts with the most lines of code and they are only reused twice. The number is worse in *SweetSpots* because of the contribution of the *Follow* concept, which is not used in any other application. However, the follow concept implements more functionality than what we use in *SweetSpots*. In *SweetSpots*, we use *Follow* only partially as a way for allowing users to save favorite spots. The *Follow* concept provides, in addition to letting sources follow a target, functionality for targets to post messages and has components to get all messages of targets a sources follow.

### **Total Effort Savings**

The total number of lines of code in all *Déjà Vu* implementations of the study applications is 11,566. The total number of lines of code in the concepts, adjusted by  $f_{tech}$ , is 18,183. Therefore, the total number of lines of code that were required to implement all 12 applications in *Déjà Vu* is 29,749. In contrast, the total number of lines of code in all the student implementations, adjusted by  $f_{\Delta}$ , is 56,590. In total, using *Déjà Vu* to replicate all 12 applications resulted in 26,841 less lines of code or 47.43% less code.

Note how with only 12 applications, we are already past the effort break-even point, with an 47.43% savings in total and per-application effort savings on all 12 applications. If more applications of the same kind are developed, the improved concept reuse would deliver even further gains. If we ignore the code of the concept implementations, using Déjà Vu to implement the 12 applications resulted in 79.56% less code.

## 8.7 Quality Analysis

In this section, we discuss and compare the quality of the Déjà Vu and the student implementations. We focus our analysis on usability (§8.7.1) and briefly discuss performance (§8.7.3), security (§8.7.2), and other quality attributes (§8.7.4).

### 8.7.1 Usability

ISO/IEC 29110 [27] defines usability as the degree to which an application can be used by users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use. In general, according to an informal heuristic evaluation we conducted on the student and the Déjà Vu applications, we have found Déjà Vu applications to be more usable than the student implementations. Needless to say, the students only had a limited amount of time to implement their application and, with enough time, could have detected and fixed many of the issues discussed below. But we think that the way Déjà Vu applications are built may reduce the likelihood of making the following kinds of usability errors: having a concept that doesn't satisfy usability and behavioral principles, having a concept with an internal inconsistency, partially implementing a concept, and coupling concepts.

#### **Anomalous Concept**

We have found several examples of concept implementations that don't satisfy well-known usability and software behavior guidelines. Take, for example, the concept of authentication that lets users register and sign in. It is good practice for applications

(a) *MapCampus*

(b) *Phoenix*

Figure 8-1: An error in the implementation of the authentication concept: special symbols are not allowed, which forces the user to choose a less secure password

to enforce minimum password strength requirements. For example, the National Institute of Standards and Technology (NIST) recommends setting a minimum of 8 characters and allowing, but not necessarily requiring, the use of special characters [21]. However, in the student implementation of *Potluck*, *EasyPick* and *Chorestar*, there's no minimum password length requirement. A user can, if they choose so, register an account with a one-letter password. In *MapCampus* and *Phoenix*, users are not allowed to use special symbols in their passwords (Fig. 8-1), which forces them to choose a less secure password.

There are also other usability problems in the student implementations of the authentication concept. For example, usability guidelines state that error messages should provide constructive advice on how to fix the problem and that error messages should be visible.<sup>15</sup> For example, it is good practice to show an error message close to the invalid input, since users look at the area with the form fields first. However, in the student implementation of *Listify*, while there appears to be a minimum length

<sup>15</sup><https://www.nngroup.com/articles/error-message-guidelines/>

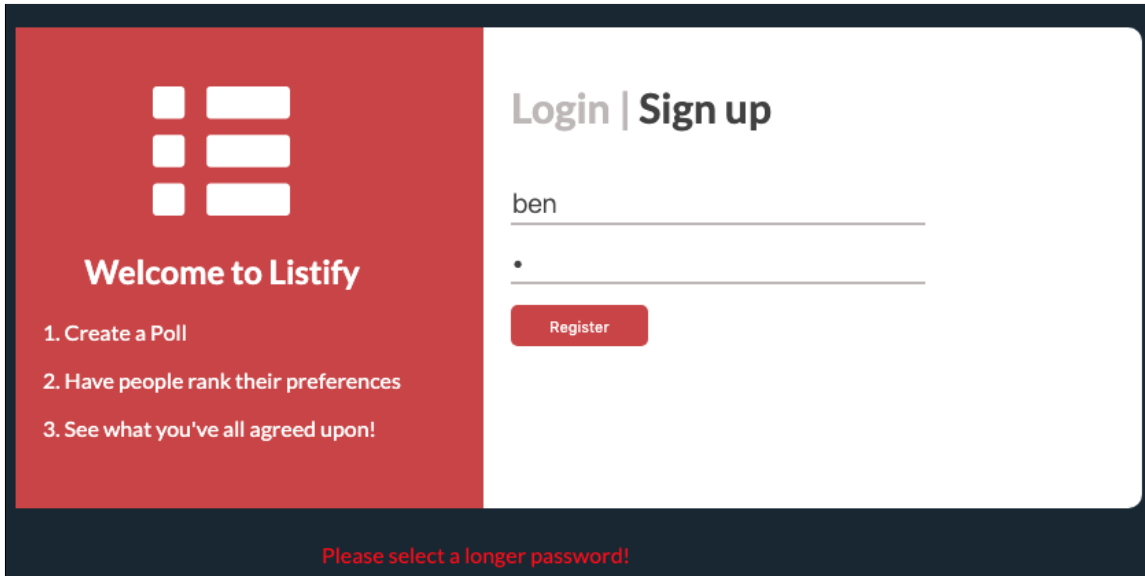


Figure 8-2: An error in the implementation of the authentication concept: if a user enters a short password, the error message doesn't say the minimum required length

requirement, the number is never revealed to the user: if the user inputs a password that is too short, the error message only says "please select a longer password!" (Fig. 8-2). Also, the error message appears at the bottom of the page, instead of appearing on the form where the error is. In *Phoenix*, if the user includes an invalid character in the password, the error message only says "Password has invalid characters" (Fig. 8-1) and doesn't provide any hints as to what the password requirements regarding valid/invalid characters might be.

While the typical implementation of upvoting/downvoting allows a user to upvote/downvote the same item only once, in *SweetSpots* a user can upvote/downvote the same review multiple times. As a result, a malicious user can boost or destroy another user's reputation score by upvoting/downvoting a same review multiple times. This is probably not a behavior the students wanted, and instead represents a bug in the implementation of the scoring concept.

### **Internally Inconsistent Concept**

Sometimes the usability problem with a concept is not that it doesn't abide by good practices, but it is instead because of an internal inconsistency. For example, in the

(a) A climbing competition organizer can input an upper-cased code for a competition

(b) A climber or spectator can't input an upper-cased competition code and it is thus impossible for a climber or spectator to log in to a competition that has an upper-cased code

Figure 8-3: An error in the implementation of the passkey concept caused by an internal inconsistency

login page of *LiveScorecard*, the input text is lower-cased, presumably, for styling purposes (Fig. 8-3): it is impossible for the user to input uppercase text in any of the form inputs. The problem is that when a climbing competition organizer creates a climbing competition, the form allows the organizer to input a code in upper case. If the organizer does so, then no climber or spectator can log in to the climbing competition, because it is impossible to input an upper-cased code in the login page.

### Concept Coupling

Sometimes, a student implementation couples different concept actions together that would be better left separate. In *Accord* for example, there is a form that allows logged-in users to update their profile or password (Fig. 8-4). The problem is that the same form is used for two different actions: changing the password and updating

The screenshot shows a web interface for updating account settings. At the top, a teal navigation bar contains the 'Accord' logo, 'Dashboard', 'About', and 'Logged In: Ben Bitdiddle'. Below this, the main heading is 'Update Account Settings'. The form consists of five text input fields, each with a label above it: 'First Name' (containing 'Ben'), 'Last Name' (containing 'Bitdiddle'), 'Current Password' (containing 'Current Password'), 'New Password' (containing 'New Password'), and 'Confirm New Password' (containing 'Confirm New Password'). At the bottom of the form is a button labeled 'Update Account'.

Figure 8-4: A coupling of actions from two different concepts: user profiles and authentication

profile information. The form is coupling two different concepts: authentication and user profile. As a result of doing so, it is hard to guess from the user interface what would happen if the user only wants to update their first name only. Can the user leave the change password field blank? It turns out that the user can indeed leave the change password field blank if they only want to update profile information, and if they do then the password is not changed. It is only if the password field is not blank that the application attempts to change the password and show an error if, for example, the confirm new password field is blank.

In *EasyPick*, users can review a course along various dimensions, such as grading fairness and overall satisfaction. But some of these dimensions are perhaps not meant to be reviews at all, and are instead meant for collecting data on other aspects of the class, such as how many hours the student spent outside of class studying. The user interface however, shows the same user interface widget, a slider, both for giving a rating and for inputting a value (Fig. 8-5). For “outside hours”, for example, is the user supposed to input the number of hours a week they spend studying outside of class? Or is it supposed to be a rating of the after-class load? Perhaps all of these

The screenshot shows a web interface for a course review form. At the top is a blue navigation bar with the 'EasyPick' logo, search, home, and user icons, and a 'Logout' button. The main content area has a white background with a blue border. The title 'Course Review Form' is centered. Below it is a search bar containing '6.170'. The selected course is 'Software Studio'. A prompt asks the user to answer questions about the class. There are two dropdown menus for 'Term taken' and 'Year taken'. Below these are five sliders for 'Class Hours', 'Outside Hours', 'Content Difficulty', 'Grading Difficulty', and 'Overall Satisfaction'.

Figure 8-5: A potential coupling of two concepts

are meant to be reviews, and the problem is that the text labels are wrong, which would make this design error an example of an anomalous concept implementation. For example, instead of “class hours”, perhaps it should say “pace is reasonable”. Or maybe what is going on is that the implementation is coupling two different concepts: rating and data collection.

### Concept Partially Implemented

Finally, we found examples in which a concept is partially implemented and it is thus lacking some basic functionality. For example, in *Phoenix*, when two users match because they have expressed an interest in meeting each other, the application lets the user write a message to their match. You would expect the application to let users send messages back and forth with their match from within the application, but it doesn't. The message is written within the application, but it is sent via email,

and no record of the message is left in the storage of the application itself. Perhaps the students wanted to build a messaging system within the application but didn't have enough time, so they ended up with functionality that is almost a message inbox with email notifications, but not quite.

### Comparison with Déjà Vu

Concept reuse in Déjà Vu helps make usability problems caused by an anomalous or inconsistent concept implementation less likely. The fact that all applications use the same concept increases the number of end-users exposed to the same concept implementation. And once a usability problem in a concept implementation is fixed, it is available to all applications using the concept.

It is unlikely that a client-side framework or server-side library would help a developer prevent errors like the ones we mentioned. For one, preventing errors like the ones we mentioned requires both client and server-side functionality working in tandem. For example, the password policy has to be consistently applied client- and server-side. And it requires domain knowledge. For example, the reason why upvoting/downvoting should be allowed only once per user is because it is an implementation of the concept of upvoting. Many user interface tool-kits include thumbs-up or up-arrow buttons, which are commonly used for upvoting. But it is still up to the developer to implement, correctly, the domain-specific behavior of increasing an item's vote count whenever a user clicks on the upvote button, but only if the user hasn't upvoted the item yet.

In Déjà Vu, concepts are decoupled by default. The developer has to actively couple them by synchronizing server-side actions and specifying bindings to get compound behavior. Since concepts are decoupled by default, it is less likely for a developer to couple them by accident. For example, the update password functionality and update user profile functionality is implemented in different concepts. The update password functionality is part of *Authentication* and to save user profiles a developer would use *Property*. In Déjà Vu, to build a form like the one in the student implementation of *Accord*, the developer has to synchronize the `change-password` component



The screenshot shows a web interface with a teal header. The header contains the logo 'Accord' on the left, and navigation links 'Dashboard', 'About', 'Ben', 'Account', and 'Logout' on the right. The main content area is white and contains two forms. The first form is titled 'Update Profile' in teal. It has two input fields: 'First Name' with the value 'Ben' and 'Last Name' with the value 'Bitiddle'. Below these fields is a teal button labeled 'Update Profile'. The second form is titled 'Update Password' in teal. It has three input fields: 'Old Password \*', 'New Password \*', and 'Retype New Password \*'. Below these fields is a teal button labeled 'Change Password'.

Figure 8-6: Update profile in the Déjà Vu implementation of *Accord*

of *Authentication* with the `update-object` component of *Property*, include `gen-id`, and so on. The easiest thing for the developer to do is to keep them separate and not synchronize `change-password` with `update-object`. In the Déjà Vu implementation of *Accord* we kept `change-password` and `update-object` separate. As a result, two separate forms appear to the user (Fig. 8-6), and it should be more clear to the end-user that it is possible to update profile information without changing the password.

Finally, it is less likely to have a concept partially implemented in a Déjà Vu application. This is because a concept must have all the basic functionality to satisfy its purpose, and once a developer includes a concept in the application, adding concept components to the application so that the concept is fully-implemented doesn't require as much effort as building the functionality from scratch using general-purpose tools.

## 8.7.2 Security

ISO/IEC 29110 [27] defines security as the degree to which an application protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization. Common attacks on web applications, such as SQL injection, cross-site request forgery (CSRF), and cross-site scripting (XSS), are less likely in Déjà Vu applications. These attacks are less likely in Déjà Vu because they are caused by implementation errors, such as forgetting to sanitize user input. Concept reuse in Déjà Vu helps prevent these common security attacks because, all other things being equal, having more people use and test the same code helps identify implementation errors sooner.

A security problem we noticed in some Déjà Vu applications is that it is easy for a developer to forget to protect actions with the `authenticate` component of *Authentication*, especially because `authenticate` produces no visible change in the behavior of the application. In most applications, users have to sign in, after which they are redirected to some other page where they can perform operations only authenticated users are supposed to perform. For example, in *Chorestar*, after a parent signs in, the parent is redirected to the parent home, where it can create chores. During manual testing, it is easy to assume that the actions are protected, since you can only reach that page using normal traversal of the site if you are already signed in. But if the `authenticate` component is not wrapped in a `dv.tx` with the component it is supposed to protect, a malicious user can craft an HTTP request to perform the action without authenticating. In this respect, however, Déjà Vu is no worse than general-purpose tools. A developer using general-purpose tools must also remember to authenticate user actions.

## 8.7.3 Performance

In the context of web applications, performance is an indicator of how well an application meets its response time or throughput requirements [61]. In all student implementations, the server functionality is implemented as a monolith and there is

therefore no need to run a transaction between different servers. In the Déjà Vu implementations, the gateway runs a two-phase commit each time the end-user triggers a transaction component. A transaction request, compared to a normal request, has an extra cost because of the extra messages required to agree on whether to commit or abort the transaction, which are unnecessary if no synchronization between different servers is required.

The performance penalty of transactions is highly dependent on the way the developer chooses to deploy a Déjà Vu application. By default, our platform co-locates on the same physical machine the concept servers of the application and the gateway. Co-locating the concept servers and the gateway eliminates what could otherwise be a long network round-trip to send requests between the gateway and concept servers and mitigates the cost of transactions.

#### 8.7.4 Other Quality Attributes

**Availability and Scalability.** Since a compiled Déjà Vu application is a standard MongoDB-Express-Angular-Node.js application (§7.3), a developer can choose to use popular platform- and database-as-a-service providers, which have very high availability and can be configured to scale automatically as user demand increases. Our Déjà Vu implementations tend to make more HTTP requests than the student implementations. This is because what would typically be only one request in a student implementation to, for example, load all data of a page, might end up being multiple requests in Déjà Vu because each concept component would send a separate request unless the concept components are synchronized in a transaction. These extra requests increase the load on the gateway, which means that more gateway replicas are needed to support the same number of end-users than replicas of the server of the corresponding student implementation. In Chapter 10, however, we discuss potential improvements to our platform implementation to mitigate this factor.

**Reliability.** Concept reuse should help make a Déjà Vu application more reliable than an application built with general-purpose tools because more end-users and

developers are testing the same concept code, which helps detect bugs faster than if each developer builds the same concepts from scratch for each application without sharing concept code.

**Maintainability.** A Déjà Vu application should be easier to maintain than an application built with standard general-purpose tools, because a Déjà Vu application is assembled from independent concept modules. Each concept can be understood and modified in isolation from the rest of the application functionality.

## 8.8 Threats to Validity

A threat to internal validity is the fact that we, the authors of the platform, were the ones that developed the Déjà Vu implementations of the student projects. Other developers might have a harder time identifying good concepts and might end up achieving a lower level of concept reuse. And then there are standard threats to external validity from the bias introduced by the selection of the subjects of our study. The subjects of our study are 12 applications developed by students for a web programming course. Naturally, the applications are about things university students are interested in, such as organizing social events and rating classes. Also, students were time-constrained, which might have influenced the type of applications they chose to develop. As a result, the applications we replicated may not be a representative sample of the types of web applications developers want to build, and our findings may not generalize to other types of web applications.

## 8.9 Summary

We summarize the results from our study by answering the original research questions:

**RQ1.** *Is it possible to build a variety of non-trivial applications using Déjà Vu, without building non-generic concepts that are specific only to a given application?*

Our findings suggest that it is possible to build a variety of applications and achieve a good level of concept reuse. We replicated the 12 student applications with a total of 18 concepts. The median number of times a concept is used in the Déjà Vu implementations of the study applications is  $Q_2 = 3$  and the median number of times a concept is instantiated is  $Q_2 = 3.5$ . Only 5 of the 18 concepts we developed are used only once and the concepts, *Chat*, *Follow*, *Match*, *Ranking*, and *Schedule*, do not appear to be application-specific.

**RQ2.** *How does the effort required to build a Déjà Vu application compare to using standard general-purpose tools?*

In total, using Déjà Vu to replicate all 12 applications resulted in 48.13% less code than the student implementations. For all 12 applications, the estimated effort required to build the application using Déjà Vu was less than the estimated effort required to build the same application with standard general-purpose tools. Since the cost of developing a concept is shared among all the applications that use the concept, if more applications of the same kind are developed, the improved concept reuse would deliver further effort gains.

**RQ3.** *How does the quality of Déjà Vu applications compare to those applications built with standard general-purpose tools?*

We have found several usability problems in the student implementations that we believe would be less likely to occur when using Déjà Vu. For security, some common security problems caused by implementation errors, such as XSS, might be less likely in Déjà Vu because of concept reuse. But, in Déjà Vu, a developer must remember to protect actions, when applicable, to prevent a malicious user from crafting a malicious request. In terms of performance, the main difference between a Déjà Vu application and a regular web application built using standard general-purpose tools is the cost of transactions, but the cost can be mitigated by co-locating the gateway and concept servers, which our platform does by default.

THIS PAGE INTENTIONALLY LEFT BLANK

# Chapter 9

## Related Work

In this chapter, we compare our approach to web application development to prior work. Our approach contributes a new type of code unit or module—a concept—and a new mechanism to put these units together—by sharing identifiers and synchronizing actions. We start by reviewing programming paradigms (§9.1) and architectural patterns (§9.2) that are primarily concerned with the way that code is organized into units of functionality, which can then be combined together through some composition mechanism. Then, we revisit the conventional approaches to web application development (§9.3) and discuss other related work (§9.4).

### 9.1 Programming Paradigms

#### 9.1.1 Object-Oriented Programming

Concepts and concept instances are roughly analogous to classes and objects. But the components of a concept, unlike the methods of an object, have full-stack implementations that include visual representations and interactive widgets.

Composing two concepts could be seen as including the behavior associated with one concept in the other and vice versa, and thus has some similarities to mixins [6, 17] and traits [59, 14]. But in *Déjà Vu*, the extra behavior being included is not necessarily orthogonal to the existing behavior since synchronizing components

intertwines these behaviors: running a concept action might also trigger some of the included behavior.

### 9.1.2 Subject-Oriented Programming

In subject-oriented programming [24], a subject is a collection of states and behaviors reflecting a particular view on a shared object. Each subject can separately define and operate upon a shared object, without any subject being aware of the state and behavior associated to the shared object by other subjects.

A subject is like a concept, but subject-oriented programming is concerned with the decomposition of the same object into different subjects. In *Déjà Vu*, concepts are, a priori, not talking about the same entities at all. It is only after they are composed together that one can see concepts as providing different views over the same entities connected by bindings.

In subject-oriented programming, subjects are composed via a composition rule, which can specify arbitrary requirements for the composition, and require the implementation of a subject compositor. The subject compositor combines subjects in an environment according to the rules. Adding new subjects to a composition requires adding new rules and modifying the subject compositor. In *Déjà Vu*, the developer determines the composition rule and the subject compositor of each application component by deciding whether the application component is a transaction or regular component, and by the property bindings.

### 9.1.3 Aspect-Oriented Programming

In aspect-oriented programming [31], the goal is to increase modularity by allowing the separation of cross-cutting concerns such as logging. The approach is to separate a program into core concerns that implement the basic functionality of the software, and cross-cutting concerns, called aspects, that encapsulate functionality that is shared by multiple core concerns. Aspects alter the behavior of core concerns by applying additional behavior, called advice, at various points in the program called join points.



Viewed through an aspect-oriented programming lens, the concepts of a Déjà Vu application are usually all core concerns. If there are join points, they would be implicit in the synchronization of the transaction components.

#### **9.1.4 Feature-Oriented Programming**

Feature-oriented programming [3] is a programming paradigm for developing software product lines. A software product line is a set of software systems that share a common, managed set of features that satisfy the needs of a particular market segment and that are developed from a common set of core assets in a prescribed way [66].

In spirit, software product line development tools are similar to Déjà Vu because, in both cases, the developer assembles an application by combining pre-built software assets. The difference is what the assets are and how they are put together. In Déjà Vu, the software assets are concepts that are combined by declarative synchronization. In feature-oriented programming, the software assets are features that are added to a program in a predefined way.

While a concept can be viewed as a kind of feature, not every feature is a concept. A feature may represent an entire collection of concepts. For example, the news feed feature on Facebook includes concepts such as feed, comment, and likes. Or a feature may represent a small increment of functionality that would be part of a concept. For example, a password recovery feature, which would be part of the authentication concept. Also, features in feature-oriented programming do not generally exist independently of the base, and are included in a predefined way. For example, in AHEAD [5], features are nested tuples of program deltas. When applied to a program, the source code is transformed by applying the delta.

#### **9.1.5 Event-Driven Programming**

In event-driven programming [20], software components can publish or subscribe to events, and the flow of the program is determined by these events. New software components can add behavior to a system by subscribing to a particular event, without

requiring the modification of the software component that publishes it. Our implementation of Déjà Vu is event-driven: concept components announce eval/exec events and are notified when it is time for them to eval/exec. But this is hidden from the application developer; application components can't announce events or have concept components listen to arbitrary events.

### 9.1.6 Postmodern Programming

Our approach could be seen as an instance of postmodern programming [65, 45, 46] in that the programming effort involves primarily gluing existing parts together rather than creating new ones. Contrary to other postmodern approaches, however, our composition mechanism and language are homogeneous. The heterogeneity of component implementations is encapsulated and not visible to the developer.

### 9.1.7 Behavioral Programming

In behavioral programming [23], an application consists of independent modules that run in parallel and communicate via events. Modules in behavioral programming, called behaviors, encapsulate a software scenario. A software scenario describes an example of how one or more users interact with the application.

Behaviors are more granular than concepts, because there's one behavior per software scenario, while a single concept would support multiple scenarios. In behavioral programming and Déjà Vu, modules can trigger, subscribe to, and block events. In Déjà Vu, however, blocking an event is not something the developer can specify as a means of controlling the flow of the program, but happens automatically when the server-side action of a component fails.

Another difference is that in Déjà Vu each concept stands on its own: you can combine a concept with another concept, but you can't change the behavior of a concept. Concepts always retain their core behavior and key properties. Behaviors in behavioral programming are less isolated: all behaviors write to the same global state and a behavior can block the execution of another behavior.

## 9.2 Architectural Patterns

### 9.2.1 Microservices

Microservices is a popular architectural style that structures an application as a collection of loosely-coupled software services that are independently deployable [44]. Microservices is an approach to service-oriented architecture (SOA) [15] that emphasizes building services around business capabilities and using lightweight communication mechanisms like HTTP.

A business capability usually involves more than one concept. Therefore, a microservice tends to aggregate more functionality than a concept. For example, an e-commerce site using microservices might have a customer feedback service that aggregates together reviews and ratings, while in *Déjà Vu*, reviews and ratings would be separate concepts.

Another difference is that, in practice, microservices provide back-end functionality only. Even in those cases in which microservices are full-stack,<sup>1,2</sup> developers have to write complex code to coordinate between different services. In *Déjà Vu*, a developer has only to specify what actions need to occur in a transaction, and *Déjà Vu* will take care of coordinating between the different concept back-ends.

*Déjà Vu* can thus be viewed as an attempt to realize a microservices architecture with full-stack microservices that are more granular, easier to combine, and generic enough to be reused in multiple applications or in a single application multiple times.

### 9.2.2 Entity-Component-System

Entity-component-system (ECS) [1] is an architectural pattern used in the development of computer games and other real-time interactive systems. In ECS, the raw data of one entity is partitioned into multiple data units called components. Each component encapsulates the data of only one aspect of the entity. The code that im-

---

<sup>1</sup><https://micro-frontends.org/>

<sup>2</sup><http://scs-architecture.org/>

plements certain functionality is located in a system, which can operate on multiple components.

A concept component is like an ECS system in the sense that it implements functionality that operates on the raw data of an entity. But, unlike a system, which can operate on multiple aspects of the entity at the same time, a concept component can only interact with one aspect of the entity. This is because a concept can't communicate with other concepts. An application component is perhaps more like an ECS system because an application component can operate on more than one aspect of the entity. But an application component, unlike a system, can't operate on the raw data of one aspect directly—it can only do so through a concept component.

## 9.3 Web Application Development

### 9.3.1 Content Management Systems

While plug-ins in content management systems are full-stack like concepts, getting different plug-ins to work together can require the developer to write complex server-side code. Concepts in *Déjà Vu*, on the other hand, can be composed declaratively in HTML. Another difference is that plug-ins tend to be more coarse than concepts. For example, a commenting plug-in might include user authentication functionality so that users authenticate before creating comments. In *Déjà Vu*, authentication and commenting functionality is implemented as different concepts.

*Déjà Vu* has no built-in support for content management, but one could develop a set of concepts that would allow a developer to create an application with a separate administrator page for managing content creation. The developer would have to combine these concepts to build their own content management system, which would be more work than using an off-the-shelf content management system, but could allow the developer to implement custom workflows or include other functionality.

### 9.3.2 End-User Development Tools

End-user development tools typically allow the user to write custom data queries and behavior, which our platform does not offer. But with an end-user development tool, a developer has to write most of the end-user behavior. In Déjà Vu, a developer can include a concept to quickly add a lot of end-user functionality to the application.

Popular tools for teaching programming, such as App Inventor [67] and Scratch [54], provide a graphical environment for building applications using a blocks-based programming language. As with end-user development tools, users can specify arbitrary behavior that might be impossible to specify in Déjà Vu, but, in contrast to Déjà Vu, users still have to write all the code for the application logic and implement each concept anew. Also, since the goal of App Inventor and Scratch is to teach computational thinking, building applications with these tools ought to resemble conventional programming. Teaching programming is not a goal of Déjà Vu and building applications with Déjà Vu bears little resemblance to conventional programming.

### 9.3.3 Web Frameworks

There are many software frameworks and libraries to support web development. Some frameworks and libraries, such as Angular, React and Vue, focus on client-side programming. Others libraries and frameworks, such as Rails<sup>3</sup> and Django<sup>4</sup>, focus on server-side programming instead. Full-stack frameworks, such as Meteor<sup>5</sup>, and tierless web programming languages, such as Links [10] and Ur/Web [8], provide support for both.

The essential difference between web frameworks and Déjà Vu, is that web frameworks are designed to be general-purpose and require the developer to write all the logic of end-user behavior. The benefit is that any application can be built using web frameworks and not just what can be built with the catalog. The drawback is that each concept has to be implemented anew. Even if full-stack components imple-

---

<sup>3</sup><https://rubyonrails.org/>

<sup>4</sup><https://www.djangoproject.com/>

<sup>5</sup><https://www.meteor.com/>

menting a particular concept such as those provided by Disqus for comments exist, or if some elements of a concept implementation can be obtained from a library, a developer still has to write complex client- and server-side code to fill in the missing parts, or to integrate the functionality with the rest of the application.

Our template language is, by design, similar to popular template languages like the Angular template language or React’s JSX. For example, components are included by name as if they were standard HTML elements; and the user can bind an expression to an input, which will recompute and update the target input property when data changes. The difference lies in the fact that *Déjà Vu* components have an eval and exec action, and that there are different types of components, transaction and non-transaction components, that the developer can create to determine behavior without having to write any JavaScript.

## 9.4 Other Related Work

### 9.4.1 Design Patterns

Concepts provide a recipe for implementing a solution that satisfies a particular purpose. In this sense, they are related to Alexander’s design patterns [2], analysis patterns [18], or the more implementation-centric patterns of object-oriented programming [19]. Unlike these, concepts not only describe a solution but they make it tangible: they embody the design pattern, and can be readily executed and combined.

The programmer’s apprentice project [56] includes a catalog of commonly recurring structures in code, requirements, or other phases of software development. These structures, like our concepts, capture and implement a common pattern. But our concepts capture higher-level aggregations of behavior.

### 9.4.2 Federated Databases

Much work has been done in the database community on federated databases [60, 25] that aim to map multiple autonomous database systems into one. Within a federated

database, a single query can access or mutate data that is distributed among multiple database systems.

While one could regard a concept as a database with queries and an application as a federated database, in federated databases, the databases are talking about the same underlying entities; it is the representation that's different. The purpose of joining them is to have a complete view of those entities. In *Déjà Vu*, the entities of concepts are different projections of behavior satisfying different purposes, which are then bound together to achieve the effect of a domain-specific entity. Also, concept components, unlike queries, have a visual representation.

Schema and ontology matching [53, 30, 9] focus on the problem of automatically discovering a correct mapping between two schemas or ontologies. All these efforts have a different purpose and are orthogonal to our work, since we are not concerned with automatically discovering bindings. Instead, in *Déjà Vu* the bindings are provided by the developer to shape the behavior.

## 9.5 Summary

Other code units proposed by programming paradigms and architectural patterns, whether they are objects in object-oriented programming or subjects in subject-oriented programming, encapsulate a different kind of behavior and have a different granularity than concepts. Closest to concepts are full-stack microservices, but to integrate full-stack microservices the developer has to write complex code to integrate different back-end services. In *Déjà Vu*, concepts are composed declaratively in HTML and our runtime system automatically coordinates with concept servers to run a transaction if necessary. Our mechanism of composing concepts by synchronizing actions is perhaps closest to the way behaviors are composed in behavioral programming. But in *Déjà Vu*, a developer cannot block events directly as a way to specify control flow. Since concepts encapsulate larger aggregations of behavior, it is not necessary for the *Déjà Vu* developer to have such fine-grained control on the flow of the program.

THIS PAGE INTENTIONALLY LEFT BLANK



# Chapter 10

## Discussion

In this chapter, we propose possible directions for improving the applicability and effectiveness of our approach (§10.1). We then discuss open questions (§10.2) and conclude the thesis (§10.3).

### 10.1 Future Directions

#### 10.1.1 Platform Improvements

There are several improvements to the implementation of our platform that would make it easier for programmers to author applications and concepts; and other improvements we could implement to improve the performance of Déjà Vu applications.

##### **Detecting Programming Errors**

**Type Errors.** We could incorporate a static type system to Déjà Vu that would enable our platform to detect a large class of errors at compile-time, and warn the programmer accordingly. The inclusion of such system would have no impact on the Déjà Vu programmer at all, who would still be able to write expressions without having to include type annotations. Only changes to concept components would be required. Concept components would now have to declare a type for each input and output property. Since we use TypeScript to author components, most inputs and

outputs have type annotations already. Déjà Vu could then detect at compile time if the type of a value produced by an expression doesn't match the type of the input the expression is bound to, or if an operator in a template expression is applied to a value of the wrong type.

**Deadlocks.** We could also perhaps incorporate something like session types [26] to detect statically if a required input was not given, or if there is a deadlock in an application component. In Déjà Vu, it is possible to write code that causes the application to freeze. This could happen if, for example, the programmer forgets an input/output binding, or if the output of a component that is produced after a transaction is bound to a required input of another component in the same transaction. Writing code that freezes an application is perhaps more likely to happen in Déjà Vu than in application development in general, because actions can block when they are triggered externally and don't have the inputs they need to run. Incorporating something like session types would allow the platform to warn the programmer if the application will freeze at run time. This would require changes in concept components, since they would now have to specify what inputs are required and what outputs are produced as a result of what actions, but could make Déjà Vu programmers more productive.

**Missing Authentication Checks.** To prevent the programmer from forgetting to add the `authenticate` component to operations that should be authenticated, we could analyze the application's source code and warn the programmer if we find some actions of a concept protected but not others. For example, in the style of [43], we could warn the programmer if we find a `create-score` action protected, but a `delete-score` not.

### **Allow Concept Authors to Use Other Front-End Frameworks**

It would be ideal to make it possible for concept authors to use React or other front-end frameworks or libraries other than Angular to develop components. This might require a change in the runtime system so that the client-side synchronization

and input binding is not built atop Angular, or it might require us to find a way to wrap components developed using other frameworks or libraries with an Angular component.

### **Make It Easier to Integrate Third-Party APIs**

Currently, for creating a concept that integrates a third-party API service, such as Stripe<sup>1</sup> for payments, programmers need to create a concept that wraps the API service and implements transactions. We have yet to develop a library or, at the very least, a set of guidelines that would make it easier to develop these wrappers.

### **Performance Improvements**

Regarding performance, we could minimize the number of extra requests a Déjà Vu application makes compared to an application developed using general-purpose tools. Many components run their eval action to get data from their servers as soon as they load. As a result, when a page first loads, many requests might be sent to the gateway, at almost the same time. To minimize the load on the gateway, we could have the client-side library wait until all components finish loading and batch all eval requests into one compound request, which is then demultiplexed server-side by the gateway.

To improve the performance of transactions, we could implement popular optimizations to the two-phase commit protocol [58, 33], which would make the two-phase commit code harder to implement in the gateway and concept servers but should deliver performance gains.

#### **10.1.2 Easy Concept Authoring**

In the current implementation of our platform, there is a big difference between the effort and expertise required for assembling applications from predefined concepts and building a concept. Building a new concept or modifying an existing concept is as

---

<sup>1</sup><https://stripe.com/>

complicated as building a web application using general-purpose tools. A programmer has to deal with APIs, databases, and so on.

We can't assume every programmer will always be able to find all the functionality they need for their application in the catalog already, especially during the early stages of the platform when the catalog is under development. It is therefore important for the success of our platform to make it as easy as possible to author concepts. When it comes to concept building, our efforts have focused on (1) developing tools and various libraries so that building a concept is not much harder than building a conventional web application using general-purpose tools, and (2) making the platform as agnostic as possible to the technologies chosen by the programmer to build a concept. But there is more that we could do to make it easier for programmers to author or modify concepts.

First, we could make it easier for programmers to modify an existing concept. A programmer might want to modify an existing concept by changing the code of an existing component or server-side action, or the programmer might want to add a new component or server-side action. Currently, any of these changes require the programmer to develop a new concept by copying the existing concept and making the code modifications. It should be possible to develop a mechanism for the programmer to be able to write the code for a new component or server-side action, mark the new code as belonging to a concept in the catalog and specify whether it should override an existing component or action. Then the platform could apply the changes to the concept. This is just like mixins in object-oriented programming [6, 17], but the units of behavior being mixed-in, in this case, are components and server-side actions instead of methods.

Finally, we could make it easier for end-user programming tools to integrate with our platform so that programmers can use an end-user programming tool to build or modify a concept, and then have the end-user programming tool export a component, a server-side action, or perhaps even a whole concept that could be used in *Déjà Vu*.

### 10.1.3 A Graphical Environment

#### Motivation

While some programmers, especially those already familiar with HTML and CSS, will prefer to develop Déjà Vu applications using the HTML-based language and their favorite editor, we expect many other programmers to prefer a graphical environment. After all, a graphical environment can, for example, prevent syntax errors by making it impossible to create syntactically incorrect code, it can help with recall by allowing the programmer to see concept and component information within the environment, it can eliminate the edit-compile-debug cycle by giving the programmer immediate feedback on the behavior of the application as it is edited [63], and it can present visualizations that allow the programmer to better understand the behavior of the application being developed. Moreover, Déjà Vu programming is well-suited for a graphical environment because the configuration and composition of concepts is amenable to graphical representations since concept components are graphical user interface elements.

Together with the development of the platform, we have been developing such an environment [41]. In this section, we summarize the work we have done on the graphical environment so far and discuss future work.

#### Building Applications with the Déjà Vu Designer

The Déjà Vu designer is a desktop software application that allows a programmer to build a Déjà Vu application graphically. In the designer, programmers can include and configure concepts (Fig. 10-1), create new application components via drag-and-drop, and bind inputs/outputs. To specify an input/output binding, the programmer can input an expression in the input field or can drag-and-drop an output of a component into the input field, which would automatically populate the input with the corresponding expression code that uses the dragged output value (Fig. 10-2).

To help the programmer understand data flows in a component, the designer can overlay input/output information next to child components. The input/output hints

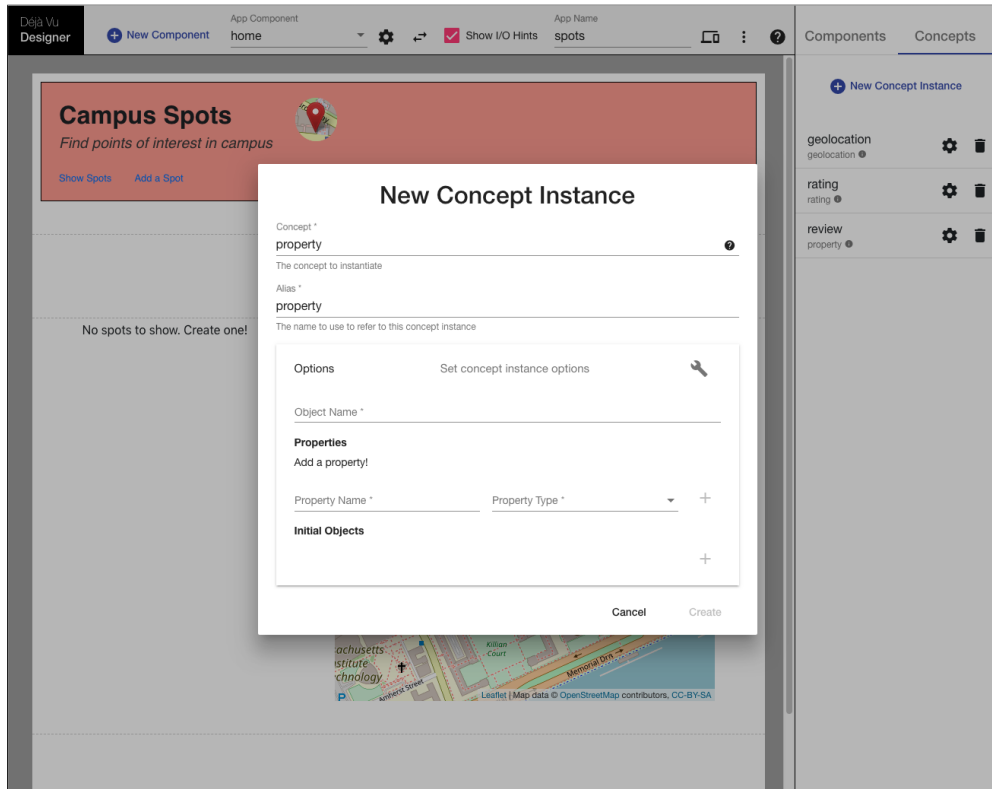


Figure 10-1: Including and configuring concepts

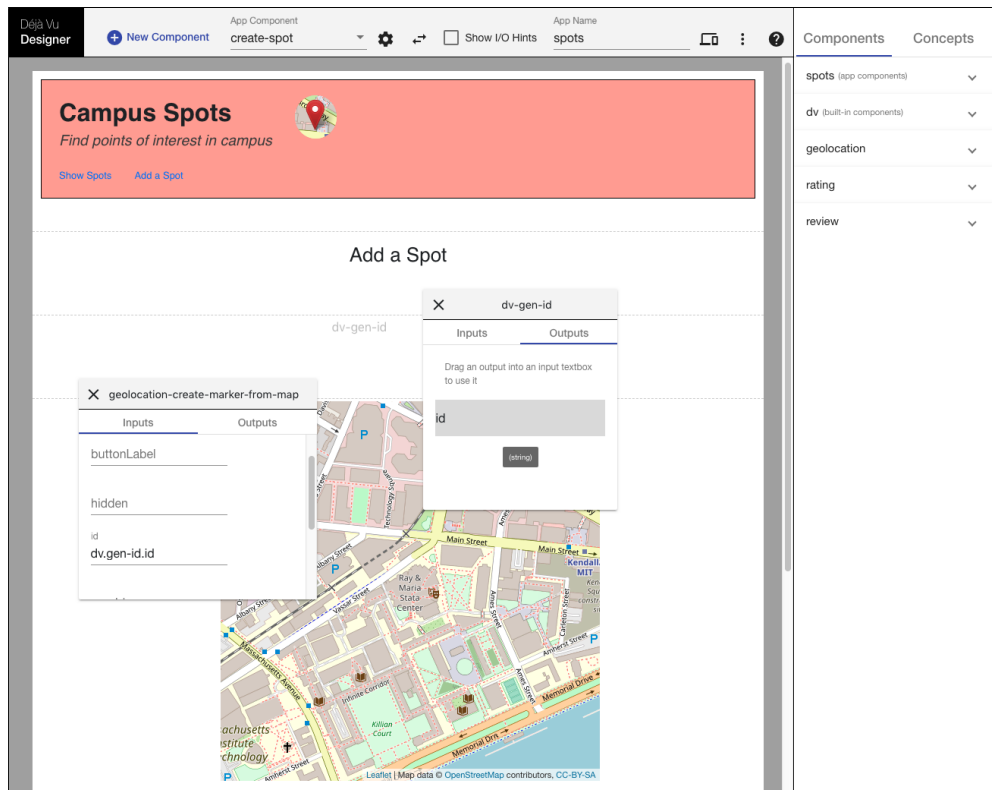


Figure 10-2: Input/Output binding

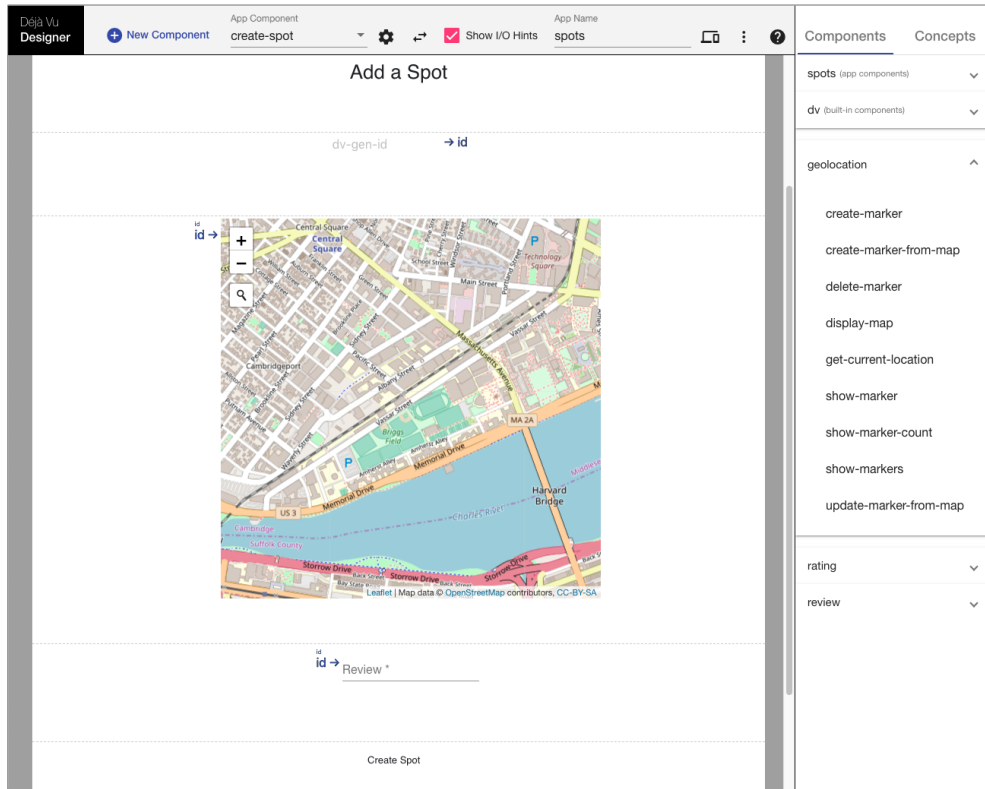


Figure 10-3: Input/Output hints

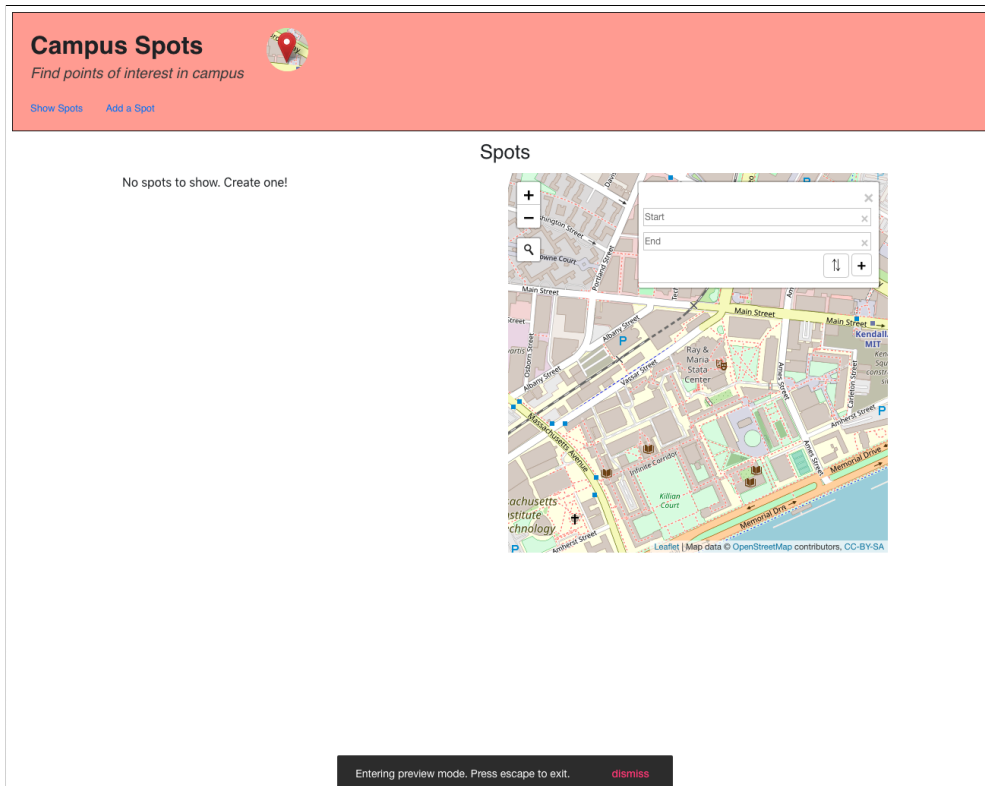


Figure 10-4: Preview mode

appear next to components that have an output property that is used somewhere else in the application component, or that have an input bound to an expression that consumes an output from some other component (Fig. 10-3). Input/output hints are color-coded by output.

The application under development is always running in the background. A programmer can interact with the application through the component currently being edited on the canvas, or the programmer can switch to a preview mode that hides the designer controls and shows the application as it would look like to an end-user (Fig. 10-4). A Déjà Vu Designer application can be saved locally or it can be exported into valid Déjà Vu application code that uses the HTML template language.

## Design Decisions

There are three key ideas in the design of our graphical environment:

- *Row-based layout model.* The component canvas is broken down into rows that contain one or more components. For arranging the layout of components within a row, the Déjà Vu Designer leverages CSS flexbox<sup>2</sup>. Flexbox is a one-dimensional layout model that offers space distribution between items and alignment capabilities. In the Déjà Vu Designer, a programmer can specify how free horizontal space is distributed in each row, how components are aligned vertically, and whether each component should retain its initial size or stretch to fill horizontal space.

Our row-based layout model is similar to the one used in Google Sites<sup>3</sup>. But Google Sites breaks a row into 12 columns, and components are given a width in terms of the number of columns they span. We think using flexbox works better for Déjà Vu because the programmer can let concept components take as much space as they want, and focus only on specifying how to distribute surplus horizontal space and how to align components.

---

<sup>2</sup><https://www.w3.org/TR/css-flexbox-1>

<sup>3</sup><https://sites.google.com>



- *Flat presentation.* Application components can contain other application components, and this containment can be arbitrarily deep. We have made the decision to only allow a single application component to be edited at a time. Application components contained by the application component being edited are rendered but not editable. To edit a contained application component, the programmer first needs to switch the canvas view to the contained application component.

Anonymous components, which are supported by Déjà Vu’s template language, are not supported in the Déjà Vu Designer. A programmer must explicitly create a new component before being able to pass it as input to another component.

- *Overlaid input/output hints.* To display input/output hints we overlay the hints onto the application component canvas. We could have instead created a new view that shows a dataflow graph where nodes are components and edges data flow—a standard visualization used in dataflow programming [12]. Our preference for overlaying hints onto the application canvas is to retain the canvas view, which shows the spatial relationships between components and shows how each component would appear to an end-user of the application.

## Future Directions

**Debugging Support.** Our current implementation of the graphical environment gives the programmer immediate feedback on the behavior of the application as it is edited. This can help with debugging, since it makes certain programming errors more evident. However, there is an opportunity to improve debugging support in the Déjà Vu Designer by showing the state of input/output properties as the application is run, and by allowing the programmer to modify the value of a property to see the effect of doing so on the behavior of the application.

**More Editing Projections.** Our current implementation of the environment maintains internally a JSON representation of the application under development. As the

programmer interacts with the environment via drag-and-drop or by inputting an expression textually in an input box, the internal JSON representation of the application is mutated. An editor or environment that follows this structure, in which there is an abstract representation of a program that is edited through multiple projections, is commonly referred to as a projectional or structured editor<sup>4</sup>.

Our current implementation, however, barely exploits this structure, since it provides only one projection for editing a Déjà Vu application: application components can only be created visually via drag-and-drop and expressions can only be inputted textually in an input box. More projections could be developed. For example, a programmer could benefit from being able to specify application components textually using the Déjà Vu language from within the environment. And expressions could be inputted using a graphical expression builder.

**Testing and Experimentation.** So far we have been the only users of the graphical environment. Further testing and experimentation on the designer is required. A benefit of having a graphical environment is that we can conduct a user study to evaluate how easy it is for developers to use our approach to build web applications, without having to find user study participants that are familiar with HTML.

## 10.2 Open Questions

Our work to date has focused on the development and validation of our approach and on the implementation of a platform to support building applications in the new style we propose. There are some open questions regarding the nature of the concept catalog and the characteristics of applications for which Déjà Vu is well-suited for, which we hope to be able to answer if the platform gains adoption and many more applications are developed using Déjà Vu.

We previously discussed criteria for creating concepts (§7.4.2), but many questions regarding the nature of the catalog remain open. For example, how many concepts

---

<sup>4</sup><https://martinfowler.com/bliki/ProjectionalEditing.html>

are there? Hundreds, thousands, millions? Does concept usage follow a power law, where relatively few concepts are widely used and there's a long tail of barely used concepts? Or would concept usage be more uniformly distributed? Our case study results suggest that concept usage distribution might be closer to the power law, since we have 3 concepts, *Authentication*, *Authorization*, and *Property*, that are used in each one of the applications we replicated; a slightly larger set of 5 concepts, *Comment*, *Event*, *Geolocation*, *Group* and *Rating*, that are widely used; and a long tail of 10 concepts, *Chat*, *Follow*, *Label*, *Match*, *Passkey*, *Ranking*, *Schedule*, *Scoring*, *Task*, and *Transfer*, that are used much less than the others. But the concept usage distribution might change as more applications are developed—especially, if the new applications are very different from the kinds of applications we replicated in our case study.

The other set of open questions are related to the applicability of our approach. What applications are better-suited for Déjà Vu? Can we develop measures of which applications it will be more or less useful for? The metric we developed in our case study (§8.6) suggests an answer to these questions: Déjà Vu is more useful for building applications whose functionality can be easily split into multiple concepts that are generic and common enough to be reused by other applications. But it doesn't say much about what types of web applications these are. For example, there might not be much to gain from using Déjà Vu to build a web-based word processor like Google Docs, because most of the complexity of a word processor is in the editor itself, which would end up being one big concept.

Perhaps the best application of Déjà Vu is as a framework for creating software product lines. To develop a software product line in Déjà Vu one would identify and implement the relevant concepts for that software family, and then assemble each individual application from these set of predefined concepts.

## 10.3 Conclusion

This thesis described a new approach to web application development and a new platform called Déjà Vu that supports building applications in this new style. In

Déjà Vu, a programmer assembles an application from predefined concepts by writing HTML and CSS, plus a small JSON configuration file. No client- or server-side procedural code is required.

Concepts are full-stack units of behavior that encapsulate all the client and server-side code required to support a motivating end-user purpose. Concepts export a set of graphical user interface components with associated server-side actions, which the programmer can include in their application and compose with other components by synchronizing the server-side actions. To link different concept entities, a programmer includes a built-in component that generates a unique identifier and provides the generated identifier as input to the concept components on the page.

Results from a case study we conducted suggest that a variety of non-trivial applications can be built with Déjà Vu, without building application-specific concepts that cannot be reused. Moreover, we have shown that concept reuse helps prevent several kinds of usability problems, and that many applications can be built using our approach with less effort than with conventional approaches to web application development.

Building modern web applications with rich functionality is no easy task and usually requires professional developers with advanced programming knowledge and skills. We hope our platform will help programmers develop more usable applications with less effort and make web application development more accessible. At the very least, we hope to spur interest in making concept design and development a central aspect of software development. Many authors have long recognized the centrality of concepts, and the importance of conceptual integrity in product design [7, 47, 13]. Yet concept design and implementation don't yet play the central role in application development one might expect, and current languages and tools lack support for concept encapsulation and reuse. By making concepts the building blocks of applications, our platform emphasizes concept design and development, and allows programmers to reuse all the design work done to invent and discover the right concepts.

# Bibliography

- [1] T. Alatalo. An entity-component model for extensible virtual worlds. *IEEE Internet Computing*, 15(5):30–37, Sep. 2011.
- [2] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Center for Environmental Structure Berkeley, Calif: Center for Environmental Structure series. OUP USA, 1977.
- [3] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Publishing Company, Incorporated, 2013.
- [4] Maryam Archie. Creating a cliché library for social applications. Master’s thesis, Massachusetts Institute of Technology, 2019.
- [5] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE ’03, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.
- [6] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA/ECOOP ’90, pages 303–311, New York, NY, USA, 1990. ACM.
- [7] Frederic Phillips Brooks. *The Design of Design: Essays from a Computer Scientist*. Addison-Wesley Professional, 2010.
- [8] Adam Chlipala. Ur/web: A simple model for programming the web. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’15, pages 153–165, New York, NY, USA, 2015. ACM.
- [9] Namyoun Choi, Il-Yeol Song, and Hyoil Han. A survey on ontology mapping. *SIGMOD Rec.*, 35(3):34–41, September 2006.
- [10] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects*, FMCO’06, pages 266–296, Berlin, Heidelberg, 2007. Springer-Verlag.

- [11] Santiago Perez De Rosso and Daniel Jackson. Purposes, concepts, misfits, and a redesign of git. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016*, pages 292–310, New York, NY, USA, 2016. ACM.
- [12] Jack B. Dennis. First version of a data flow procedure language. In B. Robinet, editor, *Programming Symposium*, pages 362–376, Berlin, Heidelberg, 1974. Springer Berlin Heidelberg.
- [13] H. Dreyfuss. *Designing for People*. The Classic of Industrial Design. Allworth Press, 2003.
- [14] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, 2006.
- [15] Thomas Erl. *Service-oriented architecture: concepts, technology, and design*. Pearson Education India, 1900.
- [16] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, May 2002.
- [17] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '98*, pages 171–183, New York, NY, USA, 1998. ACM.
- [18] Martin Fowler. *Analysis patterns: reusable object models*. Addison-Wesley Professional, 1997.
- [19] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
- [20] David Garlan and David Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM'91 Formal Software Development Methods*, pages 31–44. Springer, 1991.
- [21] Paul A. Grassi, Elaine M. Newton, Ray A. Perlner, Andrew R. Regenscheid, William E. Burr, Justin P. Richer, Naomi B. Lefkowitz, Jamie M. Danker, Yee-Yin Choong, Kristen Greene, and Mary F. Theofanos. Digital identity guidelines – authentication and lifecycle management. Technical Report NIST SP 800-63B, National Institute of Standards and Technology, 2017.
- [22] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, UK, 1978. Springer-Verlag.
- [23] David Harel, Assaf Marron, and Gera Weiss. Behavioral programming. *Commun. ACM*, 55(7):90–100, July 2012.

- [24] William Harrison and Harold Ossher. Subject-oriented programming: A critique of pure objects. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 411–428, New York, NY, USA, 1993. ACM.
- [25] Dennis Heimbigner and Dennis McLeod. A federated architecture for information management. *ACM Trans. Inf. Syst.*, 3(3):253–278, July 1985.
- [26] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems*, pages 122–138, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.
- [27] ISO Central Secretary. Systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models. Standard ISO/IEC TR 29110-1:2016, International Organization for Standardization, Geneva, CH, 2016.
- [28] 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.
- [29] Daniel Jackson. *Design by Concept: A New Way to Think about Software*. Independently published, 2019.
- [30] Yannis Kalfoglou and Marco Schorlemmer. Ontology mapping: the state of the art. *The knowledge engineering review*, 18(01):1–31, 2003.
- [31] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. *ECOOP'97—Object-oriented programming*, pages 220–242, 1997.
- [32] Butler W. Lampson. Hints for computer system design. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, SOSP '83, pages 33–48, New York, NY, USA, 1983. ACM.
- [33] 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.
- [34] Czarina Lao. Designing cliché authorship in the déjà vu web development platform. Master's thesis, Massachusetts Institute of Technology, 2019.
- [35] Donald C Latham. Department of defense trusted computer system evaluation criteria. *DoD 5200.28-STD*, 1986.

- [36] Paul J Leach, Michael Mealling, and Rich Salz. A universally unique identifier (uuid) urn namespace. *RFC 4122*, 2005.
- [37] Henry Lieberman, Fabio Paternò, Markus Klann, and Volker Wulf. *End-User Development: An Emerging Paradigm*, pages 1–8. Springer Netherlands, Dordrecht, 2006.
- [38] Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, pages 50–59, New York, NY, USA, 1974. ACM.
- [39] Matt McCutchen, Shachar Itzhaky, and Daniel Jackson. Object spreadsheets: A new computational model for end-user development of data-centric web applications. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2016, pages 112–127, New York, NY, USA, 2016. ACM.
- [40] M. Douglas McIlroy. Mass-produced software components. In Peter Naur and Brian Randell, editors, *Software Engineering, Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968*, 1968.
- [41] Barry A McNamara III. A graphical environment for déjà vu app development. Master’s thesis, Massachusetts Institute of Technology, 2019.
- [42] Kevin Mullet and Darrell Sano. Designing visual interfaces. *Acm Sigchi Bulletin*, 28(2):82–83, 1996.
- [43] Joseph P. Near and Daniel Jackson. Finding security bugs in web applications using a catalog of access control patterns. In *Proceedings of the 38th International Conference on Software Engineering, ICSE ’16*, pages 947–958, New York, NY, USA, 2016. ACM.
- [44] Sam Newman. *Building Microservices*. O’Reilly Media, Inc., 1st edition, 2015.
- [45] James Noble and Robert Biddle. Notes on postmodern programming. In *Proceedings of the Onward Track at OOPSLA*, volume 2, pages 49–71, 2002.
- [46] James Noble and Robert Biddle. Notes on notes on postmodern programming. *SIGPLAN Not.*, 39(12):40–56, December 2004.
- [47] Donald Norman. *The Design of Everyday Things*. Basic Books, 2002.
- [48] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.
- [49] David L. Parnas. Designing software for ease of extension and contraction. In *Proceedings of the 3rd International Conference on Software Engineering, ICSE ’78*, pages 264–277, Piscataway, NJ, USA, 1978. IEEE Press.



- [50] Santiago Perez De Rosso and Daniel Jackson. What’s wrong with git? a conceptual design analysis. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 37–52, New York, NY, USA, 2013. ACM.
- [51] Santiago Perez De Rosso, Daniel Jackson, Maryam Archie, Czarina Lao, and Barry A. McNamara III. Declarative assembly of web applications from predefined concepts. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, pages 79–93, New York, NY, USA, 2019. ACM.
- [52] G. D. Plotkin. A structural approach to operational semantics, 1981.
- [53] Erhard Rahm and Philip A Bernstein. A survey of approaches to automatic schema matching. *the VLDB Journal*, 10(4):334–350, 2001.
- [54] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay S Silver, Brian Silverman, et al. Scratch: Programming for all. *Commun. Acm*, 52(11):60–67, 2009.
- [55] John C. Reynolds. The discoveries of continuations. *Lisp Symb. Comput.*, 6(3-4):233–248, November 1993.
- [56] Charles Rich and Richard C. Waters. The programmer’s apprentice: A research overview. *Computer*, 21(11):10–25, November 1988.
- [57] Clay Richardson and John R. Rymer. New development platforms emerge for customer-facing applications. Technical report, Forrester Research, Inc., 2014.
- [58] 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.
- [59] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P Black. Traits: Composable units of behaviour. In *European Conference on Object-Oriented Programming*, pages 248–274. Springer, 2003.
- [60] Amit P. Sheth and James A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv.*, 22(3):183–236, September 1990.
- [61] Connie U. Smith and Lloyd G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2002.
- [62] Kevin J. Sullivan and David Notkin. Reconciling environment integration and software evolution. *ACM Trans. Softw. Eng. Methodol.*, 1(3):229–268, July 1992.

- [63] Steven L. Tanimoto. VIVA: A visual language for image processing. *Journal of Visual Languages & Computing*, 1:127–139, 1990.
- [64] Lea Verou, Amy X. Zhang, and David R. Karger. Mavo: Creating interactive data-driven web applications by authoring html. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, UIST '16, pages 483–496, New York, NY, USA, 2016. ACM.
- [65] Larry Wall. Perl, the first postmodern computer language. *Speech at Linux World*, 1999.
- [66] David M. Weiss and Chi Tau Robert Lai. *Software Product-line Engineering: A Family-based Software Development Process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [67] David Wolber, Hal Abelson, Ellen Spertus, and Liz Looney. *App Inventor*. O'Reilly Media, Inc., 2011.