Declarative Assembly of Web Applications from Predefined Concepts

Santiago Perez De Rosso
MIT CSAIL
Cambridge, MA, USA
sperezde@csail.mit.edu

Daniel Jackson
MIT CSAIL
Cambridge, MA, USA
dnj@mit.edu

Maryam Archie
MIT CSAIL
Cambridge, MA, USA
marchie@mit.edu

Czarina Lao
MIT CSAIL
Cambridge, MA, USA
mcslao@mit.edu

Barry A. McNamara III
MIT CSAIL
Cambridge, MA, USA
barryam3@mit.edu

Abstract

A new approach to web application development is presented, 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 increment of functionality. Each concept includes both front-end and back-end functionality, and exports a collection of *components*—full-stack GUI elements, backed by application logic and database storage.

To build an app, the developer imports concepts from the catalog, tunes them to fit the application's particular needs 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 that extends HTML.

The approach has been implemented in a platform called Déjà Vu, which we outline and compare to conventional web application architectures. We describe a case study in which a collection of applications previously built as team projects for a web programming course were replicated in Déjà Vu. Preliminary results validate our hypothesis, suggesting that a variety of non-trivial applications can be built from a repository of generic concepts.

Onward! '19, October 23–24, 2019, Athens, Greece

ACM ISBN 978-1-4503-6995-4/19/10…$15.00
https://doi.org/10.1145/3359591.3359728

CCS Concepts • **Software and its engineering → Abstraction, modeling and modularity**; *Organizing principles for web applications*; *Very high level languages*; *Modules / packages*; *Designing software.*

Keywords application development, web applications, concepts, design, software design, modularity

ACM Reference Format:
Santiago Perez De Rosso, Daniel Jackson, Maryam Archie, Czarina Lao, and Barry A. McNamara III. 2019. 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! '19), October 23–24, 2019, Athens, Greece.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3359591.3359728

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, 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 paper, 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.

Our paper shows how this can be done in the context of a new platform called Déjà Vu. An application is constructed by assembling concepts that are implemented as reusable, full-stack 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 HTML-like 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 fairly complex applications can be built with little effort.

1.2 Building Web Apps: The Status Quo

Suppose you wanted a simple clone of Hacker News[1], called *Slacker News* (*SN*), in which registered users can post links, comment on posts, and upvote posts or comments. How would you build such an app?

General-Purpose Tools. You might use general-purpose programming languages, coupled with a web framework, some web libraries and a database. Even if you are an advanced web developer, it would take days to build such an app. You could perhaps save time by using full-stack components provided by a web API service. For example, you could use Disqus[2] to implement comments. If the commenting functionality were isolated from the rest of the app's functionality, this could be done with very little effort (usually by just including some JavaScript and writing a little HTML). But *SN* intertwines commenting with upvoting and posting: you want Disqus's users to be the same users that can post links, and you want users to be able to upvote comments as well as posts. Even if the API were flexible enough for you to be able to pull off such an integration, it would likely require you to open up the full-stack black box, and write server-side code. For example, you might need to create a notification hook so that your app is notified when a comment is created, so that it can be given an initial score.

Content Management Systems. Another option would be to use a content management system (CMS) such as Drupal[3] or WordPress[4]. A CMS supports the creation of websites whose functionality primarily involves reading and writing content. Many CMSs provide a large suite of plug-ins that allow a website to be extended with new features. Like our concepts, 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 CMS code. For example, you might be able to include a "comment" plug-in in your app, but if it's not designed to work with the "upvote" plug-in, you'd have to write complex glue code.

Low-Code Tools. A third option is to use a low-code development platform (e.g., Microsoft Power Apps[5]), or one of the many tools and languages for end-user programming of data-centric apps (e.g., [19, 30]). 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 (which our Déjà Vu platform does not). Code for handling the mechanics of GUI elements, data storage, server requests, etc., is provided and hidden, and this is a great help. But unlike Déjà Vu, which provides implementations of rich behavior, a low-code approach will still require each new concept to be implemented anew. Moreover, functionality 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. For example, *SN* might require posts to be recommended to users based on the preferences they expressed in their upvoting of other posts; this might be provided as a machine learning plug-in. As with CMSs, such plug-ins can only be composed in ad hoc ways, and may require complex glue code.

1.3 Déjà Vu's Approach

Concept 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 [17]. For example, a comment concept manages the creation and display of comments, and a scoring concept lets users assign scores to items. Concepts are more fine-grained than traditional plug-ins or microservices, which would likely, for example, treat all the concepts associated with user reviewing (such as scoring, comments and recommendations) as part of a single plug-in or service.

Full-Stack Modules. A Déjà Vu application is a set of concept instances that run in parallel. Each instance is a full-stack service in its own right, with front-end GUI components, a back-end server and data storage, and all the code necessary to coordinate them.[6] By default, these services run entirely independently of one another, in complete isolation.[7] They share no data and do not exchange messages

[1] https://news.ycombinator.com/
[2] https://disqus.com/
[3] https://www.drupal.org
[4] https://wordpress.com/
[5] https://powerapps.microsoft.com
[6] The back-end server and data storage might be replicated for scalability.
[7] The services can run on separate machines or be co-located with our platform's runtime system for performance.

with one other. Abstractly, a concept instance is a state machine that changes state only in response to actions issued by the user through the front-end.

Data Integration Through Shared IDs. Often, concept instances hold distinct views of objects that can be thought of as shared. For example, when a scoring concept is combined with a commenting concept, one can imagine a single object having both a scoring view (containing its scoring value) and a commenting view (containing the textual comment). But, as explained above, there is no explicit sharing of data. Instead, common identifiers are used to implicitly bind the distinct views. This is achieved using a special platform function that generates an identifier, which is then passed to the two concept instances, ensuring that the objects they hold will subsequently be associated.

The binding of identifiers between concepts is the source of polymorphism in Déjà Vu. A more conventional scheme, in which concepts had type parameters that had to be instantiated, would be less flexible and would impose a greater burden on the programmer.

Declarative Client-Side Synchronization. Each GUI component of a concept contains a front-end widget and a set of corresponding server-side actions. To connect the different concepts—making their actions occur together, and passing data from one to another—the programmer references the GUI components of multiple concepts in a client-side file, along with declarative bindings that indicate which actions of the components are synchronized, and how their inputs and outputs are related.

Action synchronization is transactional, ensuring that all or none of the concept server-side actions occur. 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.

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 their actions with the actions of other concept instances, you can ensure that certain actions only occur when they should be permitted.

1.4 Contributions

The idea that software should be built from prefabricated parts dates back at least to 1968 [20]. Since then, many mechanisms to support large-scale software reuse have been developed. What's new in this work is what the parts are—*full-stack implementations of concepts*—and how they are put together—by *declarative synchronization*.

This paper 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) `create-score` with `targetId` input "ben" and `sourceId` "alyssa"

(b) `show-target` with `id` input "ben" showing the score created in Fig. 1a and ben's total score

Figure 1. Screenshots of two components of *Scoring*

- *A new platform* that supports independent development of concepts, a simple template language for instantiating and composing them, and an infrastructure for ensuring transactional semantics.
- *A case study* in which we used the platform to build a suite of non-trivial sample applications.

2 Building Apps with Déjà Vu

To see what using Déjà Vu is like, let's consider building *Slacker News* (*SN*) (see §1.2). This section is written from the perspective of the Déjà Vu user; we talk about how to author concepts later in §3.4. To build apps with Déjà Vu, users include and configure concepts (§2.1) and create app components by linking components (§2.2). Users can also provide CSS to customize the appearance of the application.

2.1 Including and Configuring Concepts

Choosing Concepts. The process of building a Déjà Vu app begins by navigating the catalog of concepts to find the concepts that provide the functionality you need for your app. The documentation accompanying a concept includes information about the configuration options and the exported components. Fig. 1 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.

SN 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 provides a data-model-defining facility for simple CRUD behavior—to save a post's author, title, and url.

Including Concepts. The concepts used by the app are specified in the app's JSON config file (Fig. 2, lines 3-22). The `usedConcepts` object has one key-value pair per concept instance. The 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

```
{
  "name": "sn",
  "usedConcepts": {
    "authentication": {},
    "comment": {},
    "post": {
      "name": "Property",
      "config": {
        "schema": {
          "title": "Post", "type": "object",
          "properties": {
            "author": { "type": "string" },
            "title": { "type": "string" },
            "url": { "type": "string", "format": "url" }
          },
          "required": [ "author", "title", "url" ]
        }
      }
    },
    "scoreposts": { "name": "Scoring" },
    "scorecomments": { "name": "Scoring" }
  },
  "routes": [
    { "path": "", "component": "home" },
    { "path": "/login", "component": "login" },
    { "path": "/post", "component": "show-post-details" },
    ...
  ]
}
```

Figure 2. Excerpt of *SN*'s configuration file

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.

Configuring Concepts. In *SN*, we only have to configure *Property*. *Property* accepts a configuration variable (`schema`) that expects a JSON Schema[8] to describe the objects it will be saving. We use `schema` to specify the type of properties we expect our objects to have. In *SN*, our objects are "posts" (line 10), and we expect them to have an author, title, and a url (lines 11-15). The effect of this 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 user to provide a value for each one and check that the value given for the url field is a valid URL. If the user doesn't provide a value for each field, or if the value for url is invalid, `create-object` will show an error message.

Other App Configuration. In the app's config file, we also define the name (Fig. 2, line 2) and routes (lines 23-28) of our app. Each route maps a URL path to a component. A component that is accessible via URL is a page. *SN*'s homepage is the component `home` (line 24) because `path` is empty. If the user navigates to `/login`, the `login` component will be shown (line 25) and if they navigate to `/post`, the `show-post-details` component will be shown (line 26).

2.2 Linking Components

Each app component is written in a separate HTML file. Fig. 3 shows excerpts of the code for *SN*'s `submit-post` and `show-post` components, together with a screenshot of how they appear to users. Our template language looks, by design, similar to other template languages. To create an app component, users include components (§2.2.1) and synchronize them to implement the desired functionality (§2.2.2).

2.2.1 Including Components

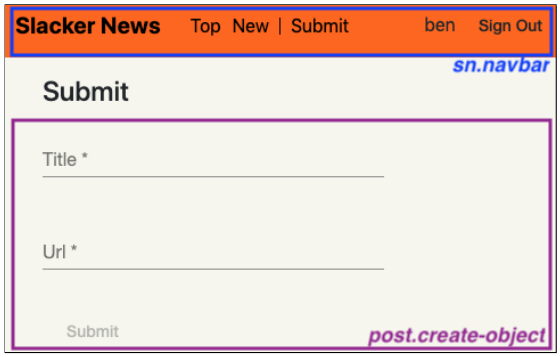
Including Components. An app component can contain other components, which can be concept components or app components. A concept component is a component that is defined and exported by a concept. An app component, on the other hand, is defined by the app developer and it is part of the app being developed. Components are included as if they were HTML elements, with the tag given by the concept instance or app name, followed by the component name. Thus, `submit-post` (Fig. 3a) includes one app component, `navbar` (line 2); two concept components, `create-object` of the "post" instance of *Property* (lines 5-8), and `create-score` of the "scoreposts" instance of *Scoring* (lines 9-11); and one built-in component, `dv.gen-id` (line 4), which generates a unique ID.

I/O Binding. Inputs to a component are bound with the syntax `property=expr`. Template expressions can include literal values, app component inputs, outputs from included components, and standard operators. 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"` on line 7, Fig. 3a, `create-object`'s button will carry the label "Submit" instead of the default button label "Create Post". The hidden property of `show-object` (Fig. 3b, line 2) indicates that the component should be activated but not visible. Thus

[8] https://json-schema.org/

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



(a) Excerpt of *SN*'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=$user />
5      <a href=post.show-object.loadedObject.url>
6        {{post.show-object.loadedObject.title}}
7      </a> ...
8      (<post.show-url showBaseUrlOnly=true
9          url=post.show-object.loadeObject.url />)
10     posted by {{post.show-object.loadedObject.author}}
11     <dv.show-date format='time-ago'
12       date=post.show-object.loadedObject.timestamp /> ...
13     <scoreposts.show-target
14       id=$id showId=false
15       showScores=false totalLabel="" /> points
16     <dv.link href="/post" params={ id: $id }>
17       comments
18     </dv.link>
19   </dv.if>
20 </dv.component>
```

(b) Excerpt of *SN*'s `show-post` component

Figure 3. Linking components

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, while the author is displayed on line 10.

App 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 app component. For example, `show-post` has an input named `id` that it uses in lines 2, 4, 14, and 16 (Fig. 3b). 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 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.

ID Sharing. To bind entities in different concepts we use a common identifier. In `submit-post` (Fig. 3a), for example, the same ID, generated by `gen-id` (line 4), is passed to `create-object` (line 5) and `create-score` (line 9). As a result, `create-score` will create a score with the same target ID as the object created by `create-object`. Similarly, in `show-post` (Fig. 3b), 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 *SN* post object.

2.2.2 Synchronizing Components

Action Types. Concept components have two 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 itself triggers the evaluation of a component, and some user interaction (e.g., 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. Note that app components don't have actions. This is because app 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 apps: 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 app components: a regular component and a transaction (tx) component. A regular component allows any of its children components

```
<dv.component name="upvote"> ...
  <dv.tx>
    <authentication.authenticate
      username=$user.username hidden=true />
    <scoringposts.create-score
      value=1 sourceId=$user.username targetId=$id ... />
  </dv.tx> ...
</dv.component>
```

Figure 4. Excerpt of *SN*'s `upvote` component that shows how we prevent unauthenticated users from upvoting posts

to eval/exec without synchronization. A tx 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 aborts. Instead of putting each component in separate HTML files, you can wrap elements in another component with the `dv.tx` tag to create an anonymous tx component with content equal to the content of the tag.

In *SN*'s `submit-post`, the tx is triggered by `create-object` (Fig. 3a, line 5) when the user clicks on the 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.

2.3 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 tx components.

For example, to implement the policy "posts must have a title" we say that the title field is required in the configuration of *Property* (Fig. 2, 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 "posts cannot be deleted", are enforced by the omission of certain components: in this case, the `delete-object` component that lets you delete *Property* objects is not included in the app. Finally, Fig. 4 shows how we use tx components to implement the policy "only authenticated users can upvote posts". In *SN*'s `upvote` component, we wrap the creation of a new score in a tx 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 app that would let a user upvote a post, only authenticated users can upvote a post.

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

3 Prototype Implementation

Déjà Vu is built using TypeScript[9], Angular[10] and Node.js[11]. The implementation consists of:

- a front-end library to synchronize components (§3.1);
- a server gateway to coordinate transactions and run security checks (§3.2);
- a compiler that transpiles a Déjà Vu app into an Angular app (§3.3); and
- a catalog of concepts (§3.4).

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

The front-end library communicates with the gateway, which then communicates with the concept servers. The communication between a concept server and its data store requires no mediation.

3.1 Front-End Library

Event Dispatching. The front-end 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 [29]: 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 app component is a tx component or not. Thus, it is as if each app component has a local mediator to coordinate its own synchronization.

Client-Server Communication. The client-side of a concept doesn't communicate with its server directly. All communication happens through the front-end library, which then communicates over HTTP to the gateway. When a component runs (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 tx, the front-end library issues an HTTP request to the gateway as soon as the run request is received. If it is part of a tx, the front-end library triggers the eval/exec of the other concept components in the app component, batches all run requests

[9] https://www.typescriptlang.org/
[10] https://angular.io
[11] https://nodejs.org

from all components that are part of the tx, 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.

3.2 Gateway Server

The communication between the gateway and concept servers happens through designated HTTP routes that all concepts are required to implement.

The gateway receives from the front-end 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 (more about this in §3.2.1). If the request is invalid, it returns an error.

If the request is valid and it is a non-tx request, the gateway forwards the request to the corresponding concept server and forwards the response obtained from the concept server back to the front-end library.

If the request is a tx request, the gateway acts as a transaction coordinator and runs a two-phase commit (2PC) protocol with all the concept servers that are part of the transaction. If all concept servers vote `ok`, the gateway commits the tx and forwards all the responses from the concept servers in one HTTP response to the front-end library. The front-end 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 concepts that voted `abort` back to the front-end library. The error responses are used client-side by concept components to show an error to the 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.

3.2.1 Security

Our runtime system has no built-in notion of authentication or authorization. This functionality is implemented in concepts, which makes it easier for 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 base, and are assumed not to have 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 app.

There are three properties of our implementation that allow us 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 their input values could be (how it knows is explained below); and (3) concepts don't communicate with their servers directly, but rather all communication is via the gateway.

On startup, our system provides the app source code to the gateway. From the source code, the gateway builds a component tree that records the hierarchical relationships between all the components in the app and the input property bindings. When the gateway receives a request to eval/exec a certain component, it verifies that the component path argument given by the front-end library is a path of the component tree. If it isn't, it returns an error.

If the component path is a valid path, it checks that the input values given to the action are valid. If, in the app's source code, the user binds a component input to a literal, and the value doesn't match the input value given to the the action, it returns an error. For example, in *SN*'s `upvote` component (Fig. 4) the user sets the score value to `1` (`value=1`, line 6). As a result, when the gateway gets a request to execute the `create-score` action in `upvote`, it checks if `value` is `1`. Thus, a malicious user can't, e.g., add `10` points to a post in one upvote (because `10≠1`).

If the request is a transaction request, the gateway additionally checks that the input values of all components that are part of the transaction are consistent: for every component of the transaction, the gateway evaluates the expression each input is bound to in the source code using the output values recorded at the time the tx was initiated, and checks that the result matches the given input values. For example, when the gateway gets a request to execute *SN*'s `submit-post` transaction (Fig. 3a), it checks that the `id` input value of `create-object` matches the `targetId` value for `create-score`. These inputs have to match because: (1) the `id` input of `create-object` is bound to `dv.gen-id.id` (line 5); (2) the `targetId` input of `create-score` is bound to `dv.gen-id.id` (line 9); and (3) they are wrapped in a `dv.tx` (lines 3-12).

3.3 Compiler

Our compiler outputs an Angular application from the app's config and component files. For each app component, it creates an Angular component. Our component language is a very thin layer atop Angular's template syntax.

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

3.4 Concept Catalog

Table 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

Table 1. Concept catalog

Concept	Purpose	# C	LoC
Authentication	Verify a user's identity with a username and password	10	1,979
Authorization	Control access to resources	12	1,929
Chat	Exchange messages in real time	5	1,185
Comment	Share reactions to items	6	1,308
Event	Schedule events	8	1,593
Follow	Receive updates from sources	13	2,180
Geolocation	Locate points of interest	8	1,719
Group	Organize members into groups so that they can be handled in aggregate	13	2,167
Label	Label items so that they can be found later	9	1,437
Match	Connect users after they both agree	8	1,592
Passkey	Verify a user's identity with a code	6	1,180
Property	Describe an object with properties that have values	12	3,924
Ranking	Rank items	5	1,037
Rating	Crowdsource evaluation of items	9	1,537
Schedule	Find a time to meet	8	2,666
Scoring	Keep track of scores	7	1,621
Task	Keep track of pieces of work to be done	13	2,106
Transfer	Transfer money or items between accounts	12	2,165

of lines of HTML, CSS, client- and server-side TypeScript code in the concept's implementation (LoC)[12].

This catalog is of course just a preliminary version. We hope expert users will grow the catalog and contribute new concepts ("new words to the vocabulary" in the terminology of [28]). To support concept development, we have built a CLI for scaffolding concepts and various libraries to ease common tasks, such as handling tx requests according to the 2PC protocol.

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 front-end library and invokes a method to register itself with the runtime system as soon as it loads. The same front-end 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 a 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 front-end framework they are most familiar with.

4 Evaluation

4.1 Comparison with Standard Approaches

Let's consider a small example component that lets users create posts with a rating. Fig. 5 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. 5a, template file, lines 2-3 and Fig. 5b, lines 18-19). We also assume that we have a client-side post service library for making requests (Fig. 5a, component file, line 6 and Fig. 5b, line 12).

Fig. 6 shows the server-side code of `submit` post. We consider two variants: as a monolith and using microservices. We assume that the server will process client requests and invoke the `savePost` server-side function with the correct

```
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 5. Two client-side code variants for `submit-post` (Angular and React)

[12]The count includes comments and blank lines. Unit tests are not counted.

```
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 6. Two server-side code variants for `submit-post` (monolith 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 7. `submit-post` component in Déjà Vu

inputs (Fig. 6a or Fig. 6b, line 1). In the monolithic back-end (Fig. 6a), we assume there are libraries for validating posts, ratings, and accessing the database. In the microservices back-end (Fig. 6b), 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. 5a, template file, line 1). In React, we subscribe to the `onChange` event generated by the post and rating components (Fig. 5b, lines 18-19) and to the `onSubmit` event of the form (Fig. 5b, line 17).
- *Aggregate data for 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. 7 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 eval/exec events, runs the component 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 app 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.

4.2 Case Study

We (the authors of this paper) built a small suite of applications that replicate the functionality of student projects in an undergraduate course. The student projects are from the Fall '16 offering of the class.

With the students' permission, we obtained access to their code repositories so we could run their applications and explore their behaviors. For each project, we developed any missing concepts in order to replicate the behavior of the original student app. We replicated only the core functionality, omitting behavioral details that are not essential to the app's working. We didn't use any of the code written by students other than some HTML to provide page content such as titles and CSS to style the appearance.

Table 2. Student projects we replicated in Déjà Vu. The symbols next to the code count indicate the front-end library used: †React v15, *Handlebars v4, **Jade v1

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

The 12 apps 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 names of the student projects we replicated, together with their purposes, are shown in Table 2. We also include the number of lines of HTML, CSS, client- and server-side JavaScript code for the student implementations.[13] 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.

Through the case study, we sought to answer two research questions:

- Is it possible to build a variety of non-trivial applications using Déjà Vu? (§4.2.1)
- Can this be done without building non-generic concepts that are specific only to a given app? (§4.2.2)

4.2.1 Experience

Selecting Concepts. Selecting a concept to provide required functionality is relatively easy, since concepts have mostly non-overlapping functionality. A possible source of overlap we have experienced is due to the flexibility of *Property*. For example, in *Chorestar*, parents have to be associated with the children accounts they create. This is so that, for example, parents can assign chores only to their children and not to other children in the system. There are two ways to do this. One way is to use *Group*, and compose it in such a way that when a parent account is created, a group with the same ID is created as well. Then, when a new child account is created, the child is added as a member of the group with the same ID as the ID of the logged in parent. The other option is to use *Property*, and configure it so that child profiles have a field `parentId`. Then, when a new child account is created, you create a child object with the `parentId` field set to the ID of the logged in parent. In a case like this, one could use either *Property* or *Group* to implement the desired behavior.

[13]The count includes comments and blank lines. Unit tests are not counted.

```
{ ...
"usedConcepts": { ...,
  "parentauthentication": { "concept": "Authentication" },
  "childauthentication": { "concept": "Authentication" }
}, ...
}
```

(a) Excerpt of the configuration file for *ChoreStar*

```
<dv.component name="parent-home"> ...
  <chorestar.parent-navbar /> ...
  <dv.tx>
    <h2>Add a New Child</h2>
    <dv.gen-id />
    <dv.status savedText="Child created" />
    <parentauthentication.authenticate
      user=chorestar.parent-navbar.user hidden=true />
    <childauthentication.register-user
      id=dv.gen-id.id signIn=false
      showOptionToSubmit=false />  ...
    <dv.button>Create Child</dv.button>
  </dv.tx>
</dv.component>
```

(b) Excerpt of `parent-home`

Figure 8. Excerpts of *ChoreStar*

With the exception of cases like this, selecting concepts is straightforward. In the future, however, as the catalog grows, selecting a concept might be harder. Especially if the catalog becomes more like a marketplace, in which there might be multiple variants of each concept. In this case, selecting a concept would be more akin to choosing a theme in a CMS such as WordPress.

Implementing Complex Behaviors. Non-trivial behavior that might, at first, seem like it would require a very specific concept, can often be implemented by combining existing concepts. For example, in *ChoreStar*, there are two kinds of users, parents and children, and only parents can create children accounts. This relatively uncommon feature was easy to implement in Déjà Vu (Fig. 8): we included *Authentication* twice (Fig. 8a), and synchronized the `register-user` component for children with the `authenticate` component of parents (Fig. 8b).

Incremental Development. With Déjà Vu, it is possible to develop an app incrementally. We found that this made it easier to evolve and debug applications as we built them. For example, in *Rendezvous*, campus events have a location and

a guest-list that only hosts can edit, but everyone else can view. In Déjà Vu, you can start with the event and location functionality by including *Event* and *Geolocation*, check that it works as you expect, then add guests with *Group*, and so on, one concept at a time.

Protecting Actions. We noticed that it is easy to forget to protect actions with `authenticate`, especially because `authenticate` produces no visible change in the behavior of the app. In most apps, 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. A potential solution to this problem is to analyze the app's source code and, in the style of [21], warn the user if we find e.g., a `create-score` action protected, but a `delete-score` not.

App Freeze. It is possible to write Déjà Vu code that causes the app to freeze. This is more likely to happen in Déjà Vu than in app development in general, because actions can block when they are triggered externally and don't have the inputs they need to run. This could happen if, for example, the user forgets an I/O binding, or if the output of a component that is produced after a tx is bound to a required input of another component in the same tx. This is less of a problem than it might at first appear to be: unlike the authentication problem discussed above, it is easy to realize that something is wrong with the app because it freezes on every test run. To help the app developer, one could perhaps incorporate something like session types [16] to detect statically if a required input was not given or if there is a deadlock.

4.2.2 Modularity Analysis

In this section, we analyze how the apps we built to replicate student projects use the catalog. The apps in the suite, together with the number of times they use a concept from the catalog, are shown in Table 3.

Discussion. The median number of concept types used per app is Q_2=6 (Q_1=5.75, Q_3=8). The median number of concept instances used per app is Q_2=9 (Q_1=7.75, Q_3=10). Most apps use roughly the same number of concept instances (σ=2.3). This is probably because all the student projects we replicated 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 apps require identification of users and preventing them from accessing content they don't own. Also, it is very common for apps 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 largest number of instances. This is because apps need an instance of *Property* for each kind of entity, and a given app 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 apps for managing human resources usually include functionality like this, so that managers and

Table 3. Concept usage in sample apps

Concept/App	Accord	Chorestar	EasyPick	GroceryShip	Lingua	Listify	LiveScorecard	MapCampus	Phoenix	Potluck	Rendezvous	SweetSpots	#Apps	#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		

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 app uses *Match* or *Schedule*, many apps, such as dating sites, have matching functionality. And many productivity apps provide functionality for scheduling meetings.

Deviations. We noticed that it becomes evident when some project deviates from what you'd expect the normal behavior of certain functionality to be. For example, in *Phoenix*, when two users match because they have expressed an interest in meeting each other, the app lets the user write a message to their match. You would expect the app to let users send messages back and forth with their match from within the app, but it doesn't. The message is written within the app, but it is sent via email, and no record of the message is left in the storage of the app itself. This behavior is not well supported by a concept. One could, of course, replicate this functionality by simply implementing a concept for sending emails, but a concept that implements a messaging system would make more sense. In *Phoenix*, one could then combine this messaging concept concept with a notification concept so as to notify users via email when there's a new message in their inbox.

Perhaps the students wanted to build a messaging system within the app but didn't have enough time, so they ended up with functionality that is almost a message inbox with email notifications, but not quite. Or perhaps this is what they intended to do in the first place. Either way, it raises an interesting research question—which we have yet to explore—about whether deviations from the norm (where "norm" is functionality that can be built by combining concepts Déjà Vu-style) represent design flaws in the application or the invention of a novel concept.

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 adding *Property*. It would be easier for app 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.

A different question is whether it is desirable for the catalog to contain multiple variants of a single concept, for example a variant of *Authentication* in which an email address it the primary identifier versus one in which a social security number is used. Put another way, is the catalog more like a marketplace (as found with WordPress themes, for example) or is it more like a polished standard library (as with java.util)? This remains to be seen.

5 Related Work

5.1 Programming Paradigms

Object-Oriented Programming (OOP). Concepts and their 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 interaction widgets.

The composition of different 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 [5, 10] and traits [8, 26]. But in Déjà Vu, the extra behavior is not necessarily orthogonal to the existing behavior. Synchronizing components intertwines these behaviors: running a concept action might also trigger some of the included behavior.

Subject-Oriented Programming (SOP). In SOP [15], a *subject* is a collection of states and behaviors reflecting a particular view. Each subject can separately define and operate upon shared objects, without any subject needing to know the details associated with those objects by other subjects.

A subject is like a concept, but SOP is concerned with the decomposition of the same object into different perceptions (subjects). In Déjà Vu, concepts are, a priori, not talking about the same objects 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.

Moreover, in SOP, subjects are composed via a *composition rule*, which can specify arbitrary requirements for the composition, and require the implementation of a *subject compositor* (code that 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 user determines the composition rule and the subject compositor by deciding whether the containing component is a tx or regular component, and by the property bindings.

Aspect-Oriented Programming (AOP). In AOP [18], the goal is to increase modularity by allowing the separation of cross-cutting concerns such as logging. The AOP approach is to separate a program into "core concerns" that implement the basic functionality of the software, and "cross-cutting concerns" (*aspects*) that encapsulate functionality that is shared by multiple core concerns. Aspects alter the behavior of core concerns by applying additional behavior (*advice*) at various points in the program (*join points*).

Viewed through an AOP lens, the concepts of a Déjà Vu app are usually all core concerns. If there are join points,

they would be implicit in the synchronization of the tx components.

Feature-Oriented Programming (FOP). In FOP, a program is a base program plus a stack of features [3]. While a concept might 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 validation feature", which would be part of the "authentication" concept.

Also, features do not generally exist independently of the base, and are included in a predefined way. For example, in AHEAD [4], features are nested tuples of program deltas. When applied to a program, the source code is transformed by applying the delta.

Event-Driven Programming. In event-driven programming [13], 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 app developer; app components can't announce events or have concept components listen to arbitrary events.

Postmodern Programming. Our approach could be seen as an instance of postmodern programming [23, 24] in that the programming effort involves primarily gluing existing parts together rather than creating new ones. Contrary to other postmodern approaches, however, it should be noted that our composition mechanism and language are homogeneous. The heterogeneity of component implementations is encapsulated and not visible to the end-user programmer.

Behavioral Programming (BP) In BP [14], an app consists of independent modules that run in parallel and communicate via events. Modules in BP (*behaviors*) are more granular than concepts, because there's usually one per software requirement, while a single concept would support multiple requirements. In BP and Déjà Vu, modules can trigger, subscribe to, and block events. In Déjà Vu, however, blocking an event is not something the user can specify as a means of controlling the flow of the program, but happens automatically when a component action fails.

5.2 Architectural Patterns

Microservices. Microservices is a popular architectural style that structures an application as a collection of loosely-coupled software services that are independently deployable [22]. Microservices is an approach to service-oriented architecture (SOA) [9] 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,[14, 15] developers have to write complex code to coordinate between different services. In Déjà Vu, the 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.

Entity-Component-System (ECS). ECS [1] is an architectural pattern used in the development of computer games and other real-time interactive systems. In ECS, an *entity* (ID) is partitioned into multiple *components* (the raw data of one aspect of the entity). The code that implements certain functionality is located in a *system*, which can operate on multiple components. A concept component is like an ECS's system in the sense that it implements certain functionality that operates on the raw data of one aspect of the entity. But, unlike a system, a concept component can only interact with one aspect of the entity (because it can't communicate with other concepts). App components are perhaps more like ECS's systems (because they can operate on more than one aspect of the entity), but an app component, unlike a system, can't operate on the raw data of one aspect directly—it can only do so through concept components.

5.3 CMSs and Low-Code Platforms

As discussed in §1.2, the plug-ins of CMSs and low-code platforms lack a generic composition mechanism. While plug-ins are full-stack like concepts, getting different plug-ins to work together can require a lot of effort.

5.4 Web Frameworks

There are many software frameworks and libraries to support web development. Some focus on the client-side development (e.g., Angular, React and Vue[16]) and others on the server

[14] https://micro-frontends.org/
[15] http://scs-architecture.org/
[16] https://vuejs.org/

side (e.g., Rails[17] and Django[18]). Full-stack frameworks (e.g., Meteor[19]) and tierless web programming languages (e.g., [6, 7, 27]) 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 you to write all the logic of end-user behavior yourself. The benefit is that any app can be built using web frameworks (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 implementing a particular concept exist (e.g., those provided by Disqus for comments), or if some elements of a concept implementation can be obtained from a library, you still have to write complex client- and server-side code to fill in the missing parts, or to integrate the functionality with the rest of the app.

Our template language is, by design, similar to popular template languages like the one used in Angular 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 (tx and non-tx) that the user can create to determine behavior—without having to write any JavaScript.

5.5 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 [11], or the more implementation-centric patterns of OOP [12]. 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 [25] 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.

6 Conclusion and Future Work

Déjà Vu is a new platform for assembling web apps by configuring and composing concepts drawn from a catalog. Our work so far has demonstrated the viability of our approach to build web applications. The concept composition mechanism is simple and requires writing no procedural code.

Moreover, the configuration and composition of concepts is amenable to graphical representations and suitable for a graphical programming environment since concept components are GUI elements. Together with the development of the platform, we have been developing such an environment. Coupled with a rich catalog, a graphical environment for Déjà Vu could allow app developers to build applications with rich behavior, without writing any binding or configuration code at all.

[17] https://rubyonrails.org/
[18] https://www.djangoproject.com/
[19] https://www.meteor.com/

Acknowledgments

Thank you to our anonymous reviewers and to our shepherd, Thomas LaToza, whose insightful critique and guidance greatly improved this paper. Thanks also to the undergraduate students that contributed to our project through MIT's UROP program: Yunyi Zhu, Shinjini Saha, John Parsons, Stacy Ho, Teddy Katz, and Eric Manzi. This research was funded in part by the International Design Center, a collaboration between MIT and SUTD (the Singapore University of Technology and Design).

References

[1] T. Alatalo. 2011. An Entity-Component Model for Extensible Virtual Worlds. *IEEE Internet Computing* 15, 5 (Sep. 2011), 30–37. https://doi.org/10.1109/MIC.2011.82

[2] C. Alexander, S. Ishikawa, and M. Silverstein. 1977. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press USA.

[3] Sven Apel, Don Batory, Christian Kstner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer Publishing Company, Incorporated.

[4] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. 2003. Scaling Step-wise Refinement. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03).* IEEE Computer Society, Washington, DC, USA, 187–197. http://dl.acm.org/citation.cfm?id=776816.776839

[5] Gilad Bracha and William Cook. 1990. Mixin-based Inheritance. In *Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications (OOPSLA/ECOOP '90).* ACM, New York, NY, USA, 303–311. https://doi.org/10.1145/97945.97982

[6] Adam Chlipala. 2015. 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).* ACM, New York, NY, USA, 153–165. https://doi.org/10.1145/2676726.2677004

[7] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2007. Links: Web Programming Without Tiers. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects (FMCO'06).* Springer-Verlag, Berlin, Heidelberg, 266–296. http://dl.acm.org/citation.cfm?id=1777707.1777724

[8] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P Black. 2006. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28, 2 (2006), 331–388.

[9] Thomas Erl. 1900. *Service-oriented architecture: concepts, technology, and design.* Pearson Education India.

[10] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. 1998. Classes and Mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98).* ACM, New York, NY, USA, 171–183. https://doi.org/10.1145/268946.268961

[11] Martin Fowler. 1997. *Analysis patterns: reusable object models.* Addison-Wesley Professional.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[13] David Garlan and David Notkin. 1991. Formalizing design spaces: Implicit invocation mechanisms. In *VDM'91 Formal Software Development Methods*. Springer, Springer Berlin Heidelberg, Berlin, Heidelberg, 31–44.

[14] David Harel, Assaf Marron, and Gera Weiss. 2012. Behavioral Programming. *Commun. ACM* 55, 7 (July 2012), 90–100. https://doi.org/10.1145/2209249.2209270

[15] William Harrison and Harold Ossher. 1993. 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)*. ACM, New York, NY, USA, 411–428. https://doi.org/10.1145/165854.165932

[16] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems*, Chris Hankin (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 122–138.

[17] Daniel Jackson. 2015. 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)*. ACM, New York, NY, USA, 282–296. https://doi.org/10.1145/2814228.2814248

[18] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *ECOOP'97 — Object-Oriented Programming*, Mehmet Akşit and Satoshi Matsuoka (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 220–242.

[19] Matt McCutchen, Shachar Itzhaky, and Daniel Jackson. 2016. 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)*. ACM, New York, NY, USA, 112–127. https://doi.org/10.1145/2986012.2986018

[20] M Douglas McIlroy. 1968. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*. 88–98.

[21] Joseph P. Near and Daniel Jackson. 2016. 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)*. ACM, New York, NY, USA, 947–958. https://doi.org/10.1145/2884781.2884836

[22] Sam Newman. 2015. *Building Microservices* (1st ed.). O'Reilly Media, Inc.

[23] James Noble and Robert Biddle. 2002. Notes on postmodern programming. In *Proceedings of the Onward Track at OOPSLA*, Vol. 2. 49–71.

[24] James Noble and Robert Biddle. 2004. Notes on Notes on Postmodern Programming. *SIGPLAN Not.* 39, 12 (Dec. 2004), 40–56. https://doi.org/10.1145/1052883.1052890

[25] Charles Rich and Richard C. Waters. 1988. The Programmer's Apprentice: A Research Overview. *Computer* 21, 11 (Nov. 1988), 10–25. https://doi.org/10.1109/2.86782

[26] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P Black. 2003. Traits: Composable units of behaviour. In *ECOOP 2003 – Object-Oriented Programming*, Luca Cardelli (Ed.). Springer, Springer Berlin Heidelberg, Berlin, Heidelberg, 248–274.

[27] Manuel Serrano and Vincent Prunet. 2016. A Glimpse of Hopjs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. ACM, New York, NY, USA, 180–192. https://doi.org/10.1145/2951913.2951916

[28] Guy L Steele. 1999. Growing a language. *Higher-Order and Symbolic Computation* 12, 3 (1999), 221–236.

[29] Kevin J. Sullivan and David Notkin. 1992. Reconciling Environment Integration and Software Evolution. *ACM Trans. Softw. Eng. Methodol.* 1, 3 (July 1992), 229–268. https://doi.org/10.1145/131736.131744

[30] Lea Verou, Amy X. Zhang, and David R. Karger. 2016. 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)*. ACM, New York, NY, USA, 483–496. https://doi.org/10.1145/2984511.2984551