CODESTRATES V2
A Development Platform for Webstrates

MARCEL BOROWSKI,† JANUS BAGER KRISTENSEN‡
ROLF BAGGE,‡ CLEMENS N. KLOKMOSE*†

December 2021

ABSTRACT

This report presents the Codestrates v2 development platform for Webstrates. Webstrates is an experimental platform to explore the vision of computational media. Codestrates v2 consists of three components: The Webstrates Package Manager, the Execution engine, and the authoring environment Cauldron. Together, the three components create a powerful development platform for Webstrates on top of Webstrates. This report summarizes the background of the project and its original motivation. We introduce the three components, explain their functionality and use, and provide details about how they are implemented. Finally, two example cases illustrate how the platform allows to create a simple todo list application and a computational notebook.

KEYWORDS: computational media, shareable dynamic media, malleable software, web development, Webstrates, Codestrates


1 INTRODUCTION

This report presents Codestrates v2, a development platform for creating collaborative web applications with Webstrates [5]. Webstrates is an implementation of the idea of computational media on the Web: software that is distributable across devices, shareable with other users, and malleable to let users modify their software and tools to their personal preferences. Codestrates v2 generalizes principles introduced in Codestrates v1 as a general purpose framework for building software on top of Webstrates. Codestrates v1 [9] is a research prototype that provides an authoring environment for Webstrates in the form of a collaborative computational notebook environment. It follows the literate computing paradigm [8] and structures code into paragraphs that can be organized in sections. Codestrates v1

* Department of Computer Science, Aarhus University, Aarhus, Denmark
{marcel.borowski,clemens}@cs.au.dk
† CAVI, Aarhus University, Aarhus, Denmark
{jbk,rolf}@cavi.au.dk
featured a notebook-based editing environment, a model for polyglot code execution, e.g., using both JavaScript and Python code in the same notebook, and a package manager [2] that enables users to easily share content and tools across different notebooks. The components introduced in this report build on top of Webstrates and aim to replace Codestrates v1 in a more generalized and robust fashion. Codestrates v2 consists of an Execution Engine, the Webstrates Package Manager and the Cauldron authoring environment (see Figure 1). Together, the three components allow users to collaboratively create web applications in Webstrates directly inside the web browser.

WEBSTRATES PACKAGE MANAGER. The Webstrates Package Manager (WPM) enables easy sharing of code between different documents. WPM takes the ideas of packages as found in Codestrates v1 [2] and generalizes the idea towards sharing arbitrary code and assets as packages between Webstrates. This decouples the WPM component and makes it independent of the rest of Codestrates v2.

EXECUTION ENGINE. The execution engine component, next, manages code execution and code editing. It generalizes the ideas of Codestrates v1 [9] by introducing the notion of code fragments: pieces of code that can be executed and modified individually. In contrast to Codestrates v1, which combined code execution framework and authoring environment in one component, the engine of Codestrates v2 is independent of an authoring environment and instead provides an API for instantiating code editors in a user interface that can be used in an authoring environment.

CAULDRON. Cauldron, lastly, is the first authoring environment we created for Codestrates v2. It allows users to author their applications right within the web browser. It provides a familiar IDE-like experience with a tree browser and tabbed editors.

![Figure 1: The software stack of Codestrates v2 consisting of Cauldron, the Execution Engine, and the Webstrates Package Manager.](image)

The remainder of this report is structured as follows: Section 2 introduces the Webstrates and Codestrates project and how they influenced the devel-
opment of Codestrates v2 and Cauldron. Section 3, Section 4, and Section 5 summarize the Webstrates Package Manager, the Execution Engine, and the features of the Cauldron authoring environment. Section 6 illustrates examples of how Codestrates v2 and Cauldron can be used to create applications. Section 7 details how the platform is implemented, and Section 8 concludes the report and hints towards possible future work.

2 BACKGROUND

This report continues the effort of the Webstrates [5] and Codestrates [9] projects to create experimental software platforms to investigate computational media [7]. The work on Webstrates is inspired by an attempt to create a modern take on Kay and Goldberg’s idea of personal dynamic media [4]. Personal dynamic media describes a software medium that is malleable and can be modified by end-users — Kay and Goldberg compare it with mediums such as “paper and clay.” We introduced the idea of shareable dynamic media [5] by articulating four key properties: shareability, being shareable between users and allowing for real-time collaboration, distributability, allowing documents to be distributed across heterogeneous devices, and computability, allowing documents to execute arbitrary code. Recently, we have been using the concept of computational media for reasons of simplicity and will continue to do so in this report. One of the characteristics of computational media is that it dissolves the traditional boundaries of what is an application and what is a document. A computational notebook is an example of computational media.

2.1 Webstrates

Webstrates [5] is a prototype platform for computational media. Webstrates is built on standard web technologies and functions as a web server. A document in Webstrates—a webstrate—can be opened in a regular web browser. A webstrate is identified by its URL. In contrast to a regular web page the DOM (Document Object Model) of a webstrate is synchronized in real-time with the Webstrates server and between all clients that access a webstrate. This synchronization includes HTML, CSS, and JavaScript that are part of a webstrate. Users can modify the JavaScript in the DOM in one client, e.g., through the developer tools of the web browser, and changes are synchronized immediately in all other clients, allowing to create collaborative applications in the Web.

Using transclusion, webstrates can be embedded into other webstrates and composed flexibly. Transclusion in Webstrates uses the iframe (inline frame) HTML element. A transcluded (child) webstrate can communicate with the transcluding (parent) webstrate and exchange information.

Webstrates, furthermore, supports a variety of APIs including assets storage, signaling between clients, or transient DOM elements or attributes that are not synchronized with the Webstrates server. Transient elements can be used to create user interfaces that should not be synchronized across multiple clients—enabling more relaxed WYSIWIS (What You See Is What I See) user interfaces.

Webstrates APIs: https://webstrates.github.io/userguide/api.html
(Retrieved November 30, 2021)
2.2 Codestrates v1

Codestrates v1 [9] is an authoring environment built on top of Webstrates. Codestrates v1 implements a GUI inside the web browser that allows to author webstrates and create applications directly in the browser (see Figure 2). A document in Codestrates v1 is called a codestrate. The authoring experience of Codestrates v1 was inspired by computational notebooks and the literate computing paradigm [8], which describes the interweaving of code with descriptive text within the same environment. While computational notebooks were usually used for data analysis and visualization purposes, Codestrates v1 allows to organize applications in “paragraphs”, e.g., JavaScript, CSS, HTML, and JSON paragraphs. Paragraphs, again, can be structured in sections (see Listing 1).

Figure 2: A screenshot of Codestrates v1. The user interface is inspired by computational notebooks and structured in sections and paragraphs (from [9]).

To support sharing of content or functionality across multiple codestrates, we introduced a package management [2], which was inspired by package managers in, e.g., Node.js or Visual Studio Code. Packages in Codestrates v1 are sections with a particular attribute (see Listing 1). When installing packages, the package management transcludes another codestrate, copies over relevant packages, and then removes the transcluded codestrate again.

2.3 Limitations of Codestrates v1

Codestrates v1 was used in multiple research projects including collaborative data visualizations [1], collaborative video editing [6], collaborative programming assignments [3], and a “computational labbook” for nanoscience [7]. Throughout these projects, the execution model of Codestrates v1 and its ability to author and use software within the same environments enabled users to use and modify applications without the need for additional software. The computational notebook user interface and the structure of content into paragraphs and sections, however, was limiting the expressivity within Codestrates as well as its loading time: Being forced into a one-dimensional
The linear notebook environment was a limitation both when authoring software, as it, for example, was not possible to open two paragraphs of code side-by-side, and when using software, as the user interface of Codestrates v1 was always loaded and thereby interfering with styling of applications and their loading time.

Codestrates v1 was implemented in itself, meaning that the implementation of new features happened inside the notebook itself. While this nicely illustrates the malleability of the medium, for Codestrates v2 we hired professional programmers for the development and they needed to be able to use their conventional tool chain (NetBeans IDE with refactoring tools, version control, etc.) to be productive. Hence, software built with Codestrates v2 is reprogrammable from within the webstrate, however, the three core components (Webstrates Package Manager, Execution Engine, and Cauldron) require external development tools to be modified.

3 WEBSTRATES PACKAGE MANAGER

The Webstrates Package Manager (WPM) is inspired by the Codestrates v1 package management system [2]. WPM allows to import and reuse code from other webstrates. Using WPM, it is possible to define a bundle of DOM elements and assets to a package and provide it with metadata including description, version number, author, and dependencies to other packages.

In contrast to the package management of Codestrates v1, WPM does not include a user interface. To use WPM, JavaScript functions have to be called directly, for instance, in scripts or using the developer console (see Listing 3).

3.1 WPM Packages

WPM packages exist in the DOM of a webstrate and are defined using custom `<wpm-package>` element. All child elements inside `<wpm-package>` are part of the package and the `id` attribute is used to uniquely identify a package (see Listing 2). Inside another custom element, the `<wpm-descriptor>`, it is
possible to define meta data of a package. The descriptor contains a serialized JSON object with the following properties:

- **description**: A description of the package.
- **dependencies**: Selectors for other packages that this package depends on.
- **assets**: Assets that the package uses. These will be copied over with the package when it is installed in another webstrate.
- **version**: The current version of the package.

```xml
<wpm-package id="myPackage">
  <wpm-descriptor>
    {
      "description": "My Package",
      "dependencies": [ "#myOtherPackage" ],
      "assets": [ "myImage.png" ],
      "version": "1"
    }
  </wpm-descriptor>
  <script>
    console.log("Hello from myPackage!");
  </script>
</wpm-package>
```

Listing 2: A simple example WPM package.

3.2 **Requiring Packages**

Installing packages in WPM is done by requiring them using the `require()` function: One webstrate serves the package, while other webstrates can require the package from it.

A package can be installed either transiently or persistently (see Listing 3). By default packages are installed transiently: the packages are copied into a transient element in the `<head>` element. In this case, packages are retrieved from the repository every time a webstrate is loaded and are therefore automatically updated once the version in the repository is changed. If a package is installed persistently, the package is copied into a given DOM target node and persisted by the Webstrates server. This can be useful, if a specific version of a package is used or users want to modify a package locally in one webstrate, e.g., to test new features. WPM will build a dependency tree so that dependencies are always installed before the dependent package.

4 **Execution Engine**

The development of the execution engine was motivated by the shortcomings of Codestrates v1 described in Section 2.3 and set out to decouple the user interface and authoring environment from the execution framework. In this section, we describe the execution engine. The execution engine is packaged with WPM and deployed on webstrates.
By separating the authoring environment from the execution engine it is possible to author the same code using different authoring environments. While developing Cauldron we, e.g., created a proof-of-concept version of a “linear editor” that displays code in cells like a computational notebook.

4.1 Code Fragments

As the naming scheme “paragraphs” in Codestrates v1 was related to its notebook-like interface, we renamed the “computational units” in a document from paragraphs to fragments. A fragment is a DOM element with the tag name code-fragment that has a data-type attribute (see Listing 4). The data type describes the type of content of a fragment, such as HTML, Markdown, LaToX, CSS, SCSS, JavaScript, TypeScript, Python, Ruby, and Lua. Fragments are hidden by default and not rendered in the browser. Content inside a fragment is always stored as plain text in the DOM and is, thus, not interpreted on its own by the browser.

Fragments can be placed anywhere in the DOM and structured arbitrarily, introducing the possibility to create deeper hierarchies (e.g., using <div> elements as folder). Besides this structural flexibility, it further allows to create different editors for editing the content of these fragments: It is now possible to view fragments both in a one-dimensional notebook representation as before, but also as a tabbed editor, similar to an IDE, or a two-dimensional canvas where fragments can be freely moved around.

4.2 Executing Code

The computational engine of Codestrates v2 manages the execution of fragments, the creation of virtual machines for executing code, as well as providing functionality to reuse code from fragments using require() — similar to modules in Node.js. Fragments with program code (JavaScript, TypeScript, Python, Ruby, Lua) can be “run” individually by the user to execute their code. If a fragment should be executed whenever the webstrate is loaded, it can be set to auto-run by adding the auto attribute to the fragment element (see Listing 4). A JSON fragment allows to store data inside a fragment (see the example in Section 6.1).

JavaScript fragments can run directly in the browser, while other programming languages will require a transpiling step. Thus, Codestrates v2 is polyglot, i.e. it allows fragments to be written in multiple programming languages. Even more, code from one fragment can be exported and imported in another.
Listing 4: Example of a JavaScript and HTML code fragment. Both fragments are set to auto-run, indicated by the auto attribute. The HTML fragment renders its content into a transient element. The code fragment is linked to the rendered element using a transient ID that is created on page load.

4.3 Enabling Markup and Styling

Fragments with markup (HTML, Markdown, LaTeX) and fragments with stylesheets (CSS, SCSS) can only be set to auto-run and not run manually. Content from markup and stylesheet fragments is rendered or preprocessed into a transient element with the class autoDom, which is linked to its respective code fragment using an ID (see the <transient> element in Listing 4). Content is synchronized from the fragment to the transient rendering so changes to the content of the fragment are automatically applied in the rendering (for markup and stylesheet fragments). By rendering HTML in a transient element opposed to changing it directly, it is possible to collaboratively edit the fragment code. This in turn, however, also means that changes to the rendered DOM elements are not synchronized back to the fragments, as this could cause conflicts (see Figure 3).

Figure 3: Data synchronization of HTML fragments between two clients: Fragment plain text content is synchronized both ways to the Webstrates server and Code Editor Instances. The rendered view is also synchronized to fragment plain text changes, however, is not synchronized back to the fragment.
Creating Editors

The execution engine also provides functionality to create code editors and view-only editors, or just views, for fragments that are synchronized with the fragment content. Listing 5 shows an example of how a code editor can be created for a fragment. The engine currently supports three types of editors that build on their respective libraries: Ace, CodeMirror, and Monaco.

Besides code editors, the engine can also create views, which render markup fragments (HTML, Markdown, LaTeX) into a read-only editor instance. The view is updated live whenever the fragment changes. The view is, for example, used in Cauldron to view a live preview of a fragment or in the Ganymede example (see Section 6.2) to display rendered Markdown fragments. If a markup fragment is set to auto-run a view will automatically and transiently be created as an immediate sibling to the fragment in the DOM (see Listing 4).

The creation of editors is part of the fragment and editor engine as it provides a basic way of editing fragments that handles the synchronization between the fragment and the editor. Authoring environments—like in our case Cauldron—can then use these editors and display them.

```javascript
let fragment = Fragment.one("#myFragment");
let editor = EditorManager.createEditor(fragment, {
  editor: MonacoEditor,
  theme: "light",
  mode: "full"
});
let editorNode = editor[0].html[0];
document.body.append(editorNode);
```

Listing 5: Creating an editor for a fragment in Codestrates v2.

5 CAULDRON

Cauldron is an authoring environment for Codestrates v2 that allows, e.g., to manage and edit fragments. In contrast to Codestrates v1, Cauldron does not follow a computational notebook interface but the interface of a normal tabbed editor, where tabs can be arranged side by side both horizontally and vertically (see Figure 4). Cauldron uses the GoldenLayout layout manager to arrange editors and other components. The Cauldron authoring environment itself can be positioned either docked (see Figure 4), full screen, or floating as a window. Like the engine, Cauldron also builds on top of WPM, it is, however, not implemented using code fragments but regular script elements.

To load Cauldron, a webstrate needs to include the WPM script and contain a bootstrap script that loads the execution engine using WPM. Cauldron does not need to be loaded through WPM until the user requests it to be opened. Only then is Cauldron installed transiently in the webstrate. Hence, the authoring environment does not clutter the memory of the browser when it is not needed.

2 Ace Editor: https://ace.c9.io/ (Retrieved November 30, 2021)
3 CodeMirror Editor: https://codemirror.net/ (Retrieved November 30, 2021)
4 Monaco Editor: https://github.com/Microsoft/monaco-editor (Retrieved November 30, 2021)
Figure 4: An overview of the Cauldron authoring environment for Codestrates v2: Cauldron in its docked layout allows to see the application (A) to the side. Fragments and assets can be browsed in the Tree Browser (B) and inspected and renamed using the Inspector (C). Tabbed editors (D) allow to display code editors in a flexible way. The integrated console (E) allows to inspect logs without leaving Cauldron. (Cauldron can also be used full screen (which would hide A) or floating (which would put A full screen and Cauldron as a floating window above it.)

5.1 Tree Browser, Inspector, and Tabbed Editor

The main purpose of Cauldron is to manage and author fragments. It provides three components that support this: the tree browser, the inspector, and tabbed editors.

The tree browser (see Figure 4B) shows an overview of the `body` element of a webstrate. Fragments inside the browser are indicated by icons for their type and can be opened by double-clicking on them. Regular DOM elements are displayed using folder icons, and transient elements are hidden in the tree browser. Using the context menu on elements, users can create new elements and fragments, move them, or delete them. It is, furthermore, possible to edit the HTML of elements that are not fragments (similar to the developer tools of a web browser). When editing the HTML of elements that are not fragments, changes are not automatically applied but instead have to be saved manually. Elements or fragments can also be dragged and dropped into the tree browser of other webstrates. The tree browser also allows viewing and uploading assets that are part of a webstrate. New assets can be uploaded using drag and drop.

Once an element, fragment, or asset is selected in the tree browser, the inspector (see Figure 4C) displays attributes of the element like its ID or class, and allows to edit this information using the input field.

The tabbed editor (see Figure 4D), finally, shows editors of fragments or HTML elements, which can be displayed side by side in a flexible way. When editing fragments, the editor also contains buttons to run or auto-run the fragment. Using a collaboration layer, the tabbed editors also show the text cursor and selections of other clients that are currently editing a fragment.
5.2 Console, Toolbar, and Package Manager

Besides the main authoring tools, Cauldron also adds additional functionality to support development. The console (see Figure 4E) shows the current console output. This, for example, allows to also see the console output on browsers that do not have developer tools to do so—for example, on web browsers on mobile devices.

The toolbar of Cauldron adds various shortcuts to Cauldron and Webstrates functionality like changing the permissions of a webstrate, creating a new webstrate, duplicating a webstrate, or inspecting the version history of a webstrate. It also allows to change the viewing mode of Cauldron (docked, full screen, floating window).

From the toolbar users can also open the Cauldron package manager. The package manager provides a user interface for the WPM component and allows to install or remove packages in a codestrate.

6 Examples

To illustrate the use of Codestrates v2 and Cauldron, we will present to examples of their use. The first example describes how to create a simple todo list application using Codestrates v2 and Cauldron, and the second example demonstrates how a computational notebook can be implemented.

6.1 Todo List Application

The first example is a simple todo list application that allows to add, mark as completed, and delete todos (see Figure 5). The example follows a simple MVC (Model-View-Controller) architecture and demonstrates its use in Codestrates v2 and Cauldron. The view consists of a single HTML fragment that renders the interface of the application, an external CSS library that is loaded using WPM, and a CSS fragment for styling (see Appendix A). A JSON fragment is used to store the data of the model and to synchronize the model with other clients. The controller, finally, is a JavaScript fragment that handles interactions with the view, and synchronizes its state with the model in the JSON fragment.

While this example is implemented using a MVC architecture, it is also possible to store the data of the todo items directly inside the DOM by using a persisted DOM element to implement the application instead of an HTML fragment. This would make the JSON fragment obsolete and instead combine view and model in the DOM.

6.2 Ganymede Computational Notebook

The second example is a Jupyter-inspired computational notebook called Ganymede, which allows to create code and markdown cells that can executed (see Figure 6). Code cells can be used to execute JavaScript code. Inside code cells the `print()` function can be used to output the result of a computation in the notebook. It is possible to run cells individually or

---

6 Todo List Example: https://demo.webstrates.net/codestrates-v2-todo-list/release/?copy (Retrieved November 30, 2021)
7 Ganymede Example: https://demo.webstrates.net/codestrates-v2-ganymede/release/?copy (Retrieved November 30, 2021)
Figure 5: The todo list example application created with Codestrates v2 and Cauldron.

all of them at once. Markdown cells can be edited by double-clicking on them, which displays a markdown editor instead of a view. In our example, we also demonstrate how WPM can be used to load external libraries like Vega-Lite\(^8\) to display visualizations.

Technically, cells in the Ganymede notebook are code fragments that are stored in the DOM. Using the fragment API, Ganymede can run individual fragments and display the output of computations. Also the editors in the notebook are generated using the editor API from Codestrates v2 — this also enables collaborative editing of cells. Our example, however, does not persist or synchronize the runtime state and outputs of the notebook. Excluding comments, Ganymede is implemented in less than 200 lines of JavaScript.

7 IMPLEMENTATION

Building on web technologies, WPM, Codestrates v2, and Cauldron are implemented using web technologies including JavaScript, HTML, and CSS. The code of all three components is available on GitHub.\(^9\) The following subsection describe each component in more detail.

7.1 Webstrates Package Manager

The Webstrates Package Manager is a single JavaScript script that implements the WPM object, which exposes functions to require packages. The script is usually located in the head element of the DOM and is executed on page load. Requiring packages from another webstrate is facilitated by using transclusion — similarly to the package manager of Codestrates v1 [2]. Using the custom tags described in Section 3, WPM scans the DOM for packages and parses their descriptor.

\(^8\) Vega-Lite: https://vega.github.io/vega-lite/ (Retrieved November 30, 2021)
\(^9\) WPM: https://github.com/Webstrates/WPM
Codestrates v2: https://github.com/Webstrates/Codestrates-v2
Cauldron: https://github.com/Webstrates/Cauldron
(Retrieved November 30, 2021)
Figure 6: The Ganymede computational notebook example created with Codestrates v2 and Cauldron.

7.2 Execution Engine

Once a webstrate with Codestrates v2 is loaded, it first executes the WPM scripts, which then in turn execute the Codestrates v2 scripts. The engine of Codestrates v2 is managing the execution of fragments. To do this, the engine first searches the DOM for code fragments and adds mutation observers, which detect added or removed fragments. On load or when a fragment is added during runtime it is set up: The fragment is parsed and its type is detected. If the fragment supports code execution and is set to auto-run it is executed, and if the fragment does not support code execution and is set to auto-run executable program code will be executed and markup or styling is rendered into a transient element. Table 1 shows the list of libraries used to execute the various fragment types.

The editors that Codestrates v2 provides for editing fragments are Ace, CodeMirror, and Monaco. Each of the editor types registers itself with the EditorManager, which manages the editor types and allows to create editors. To create an editor, a fragment first has to be selected and parsed to the EditorManager. Once created, mutation observers in the fragment update the editor when remote changes happen in the code fragment and the editor updates the code fragment code once a users edits code in the editor.
### Table 1: The libraries used for the different fragment types. (Retrieved November 30, 2021)

<table>
<thead>
<tr>
<th>Fragment</th>
<th>Library</th>
<th>Website</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaScript</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>TypeScript</td>
<td>TypeScript</td>
<td><a href="https://www.typescriptlang.org">https://www.typescriptlang.org</a></td>
</tr>
<tr>
<td>Python</td>
<td>Brython</td>
<td><a href="https://brython.info">https://brython.info</a></td>
</tr>
<tr>
<td>Ruby</td>
<td>Opal</td>
<td><a href="https://opalrb.com">https://opalrb.com</a></td>
</tr>
<tr>
<td>Lua</td>
<td>Fengari</td>
<td><a href="https://fengari.io">https://fengari.io</a></td>
</tr>
<tr>
<td>JSON</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>HTML</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Markdown</td>
<td>Showdown</td>
<td><a href="http://showdownjs.com">http://showdownjs.com</a></td>
</tr>
<tr>
<td>LaTeX</td>
<td>LaTeX.js</td>
<td><a href="https://latex.js.org">https://latex.js.org</a></td>
</tr>
<tr>
<td>CSS</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>SCSS</td>
<td>Sass</td>
<td><a href="https://sass-lang.com">https://sass-lang.com</a></td>
</tr>
</tbody>
</table>

#### 7.3 Cauldron

Opposed to the engine, Cauldron is not opened and loaded by default when a webstrate is loaded. We decided to do that to improve performance on page load. Instead, Cauldron can be opened and loaded using an *Edit* button in the top right corner of the page.

Once loaded, Cauldron will initialize its interface, load the GoldenLayout library and setup the editors. By default Cauldron will be docked on the right side. The DOM of Cauldron is placed outside the `body` element of a webstrate in a transient element. By placing Cauldron outside the body, it does not interfere with application content in the body and allows users to use regular CSS styling rules without having to take Cauldron into account.

To store the current location of Cauldron window and its layout including all currently open editors, Cauldron uses the `localStorage` property of the browser. After a page refresh or page navigation, Cauldron retrieves this configuration from the `localStorage` and re-arranges Cauldron in the same state it was before—except for scroll positions in editors, which are not stored in the current version.

#### 8 Conclusion

In this report, we presented Codestrates v2, consisting of three components for the development of applications in Webstrates: the Webstrates Package Manager (WPM), the Execution Engine, and Cauldron. While WPM allows for easy transfers of functionality across webstrates, the engine adds an execution framework for code based on code fragments, and Cauldron an authoring environment for these fragments. Together, the three components provide a powerful platform for creating web applications in Webstrates.
ACKNOWLEDGMENTS

We would like to thank Germán Leiva for his helpful feedback. This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 740548) and from Carlsbergfondet (grant agreement No CF17-0643).

REFERENCES


A TODO LIST EXAMPLE CODE

A.1 View

```html
<div id="main" style="display: none;">
  <section class="hero is-dark">
    <div class="hero-body">
      <div class="container">
        <h1 class="title">TODOS</h1>
      </div>
    </div>
  </section>
  <section class="section">
    <div class="content">
      <input class="input is-rounded" id="newTodo" type="text" placeholder="New todo">
    </div>
    <div id="todos" class="section"></div>
  </section>
</div>
```

A.2 Style

```css
body {
  text-align: center;
}
body input {
  width: 25em !important;
  text-align: center;
}
.todo {
  margin: 5px;
  cursor: pointer;
}
.todo.completed {
  text-decoration: line-through;
}
```
// Load external CSS framework uploaded as an asset.
await wpm.requireExternal("bulma.css");

// Show main element when bulma has loaded
document.querySelector("#main").style.display = "";

// Load the code fragment containing our data (#model)
// Fragment.one(selector) works like querySelector and finds the first
// matching fragment
let model = Fragment.one('#model');

let todos = document.querySelector('#todos');
let input = document.querySelector('#newTodo');

input.addEventListener("keyup", (e) => {
  if (e.key !== "Enter") return;
  let newTodo = {
    text: input.value,
    completed: false
  };;
  getData((data) => {
    data.todos.push(newTodo);
    storeData(data);
    input.value = "";
  });
});

getData(showTodosFromData);

// Here we listen for changes to the model fragment
model.registerOnFragmentChangedHandler(() => {
  getData(showTodosFromData);
});

function showTodosFromData(data) {
  if (!data || !data.todos) return;
todos.innerHTML = "";
  for (let todo of data.todos) {
    let todoDiv = document.createElement("div");
    let style;
    if (todo.completed) {
      style = "completed";
    } else {
      style = "is-info";
    }
todoDiv.innerHTML = `<span class="todo tag ${style}
    is-large">${todo.text}<button class="delete
    is-small"></button>`;
todos.append(todoDiv);
  }
}

let del = todoDiv.querySelector("button");
del.addEventListener("click", (e) => {
e.stopPropagation();
  let todoDiv = e.target.closest("div");
  deleteTodo(todoDiv);
53 });
54 
todoDiv.addEventListener("click", (e) => {
56   let todoDiv = e.target.closest("div");
57   let index = getTodoIndex(todoDiv);
58   getData(data => {
59     data.todos[index].completed = !data.todos[index].completed;
60     storeData(data);
61   });
62 });
63 }
64 }
65 function deleteTodo(todoElement) {
66   let index = getTodoIndex(todoElement);
67   getData(data => {
68     data.todos.splice(index, 1);
69     storeData(data);
70   });
71 }
72 }
73 function getTodoIndex(todoElement) {
74   let index =
75     Array.from(todoElement.parentElement.children).indexOf(todoElement);
76   return index;
77 }
78 function getData(callback) {
79   model.require().then(data => {
80     callback(data);
81   }).catch(err => {
82     console.log("Couldn't parse JSON data");
83     callback(undefined);
84   });
85 }
86 function storeData(data) {
87   model.raw = JSON.stringify(data, null, 2);
88 }

A.4 Model

JSON Fragment

{ "todos": [
  {
    "text": "Write some code",
    "completed": false
  },
  {
    "text": "Eat some food",
    "completed": false
  },
  {
    "text": "Sleep",
    "completed": false
  }
]