Introduction

This document contains detail information for a developer experience evaluation of the DCM-RUST architecture. This architecture is developed to facilitate runtime code mobility in Rust-based web applications.

Definition

Code mobility refers to the ability to dynamically shift the execution location of a piece of code at runtime. It means that some functions can shift their execution between the client side and the server side between invocations.

Overview

The general workflow of the DCM-RUST architecture is as shown in the figure below:

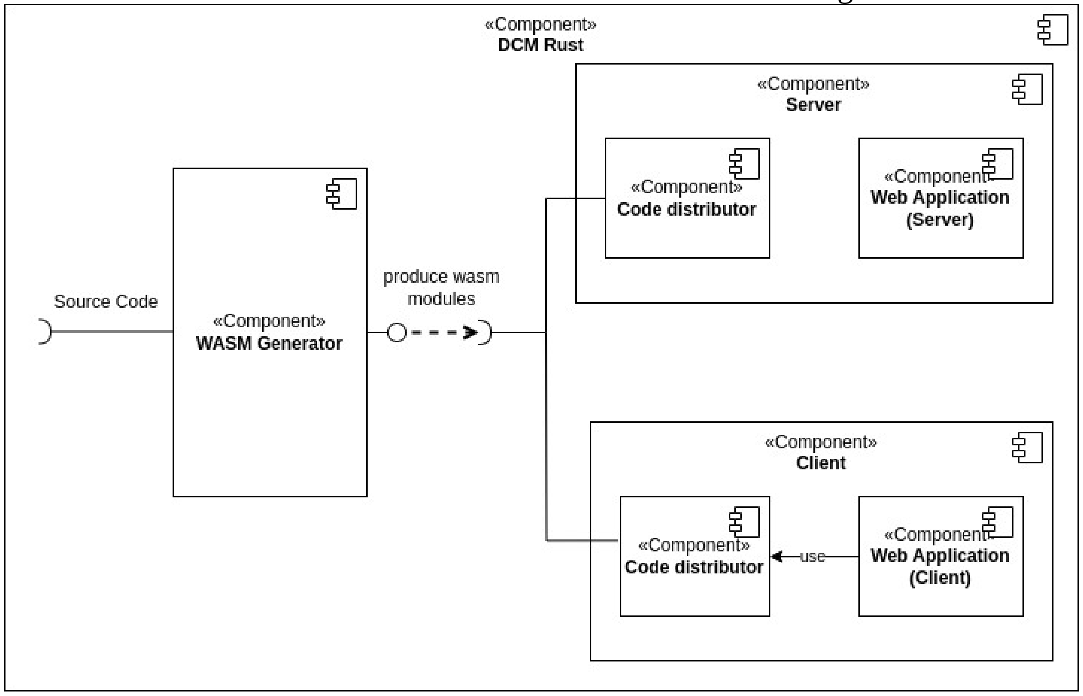


Figure 1: DCM-RUST Overview

1. The developer specifies some portions of code for mobility in their project.
2. The WASM generator infrastructure generates WebAssembly modules from the project's source code and moves them to the proper directories in client and server side code distributors.
3. Alongside, it also generates JavaScript glue code and moves them to the client side code distributor's directory.
4. Now, when the developer uses this glue code, the invocation is handled by the code distributor component, and execution happens on either the client or server side depending on the fragment's current execution location, which is decided by the decision system.
5. The decision system is not in the scope of this thesis, so, a chrome browser extension is provided to manually emulate the decision system. A simple interface is also attached to the demo-application UI as a shorthand.

The demo application

Our demo project demonstrates the overall concept in a simple way. The project is a simple online shopping application with basic features like browsing products, adding them to the cart, and placing orders. Since the decision system is not part of this thesis, an interface is provided to manually simulate the execution location of mobile functions and objects. Figure below shows the main page of the demo application. Navigate to: http://vsr-kub001.informatik.tu-chemnitz.de/ to see it live.

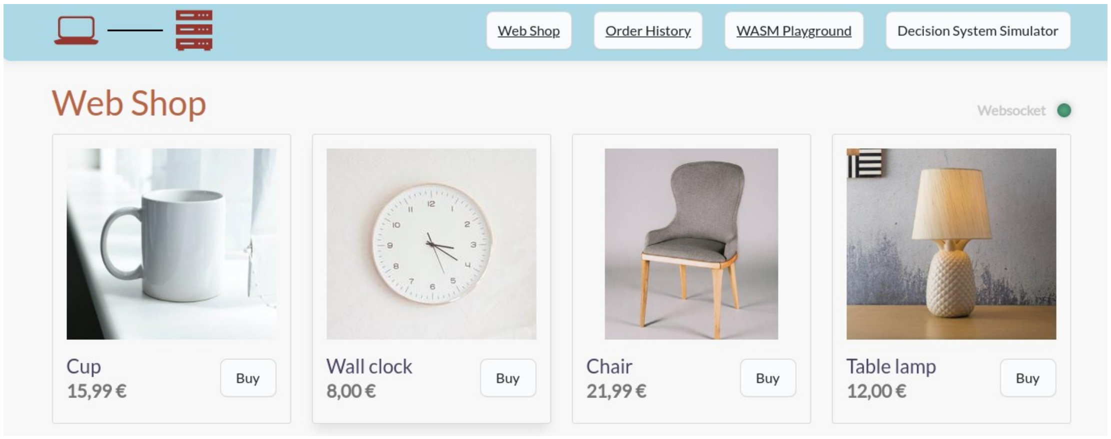
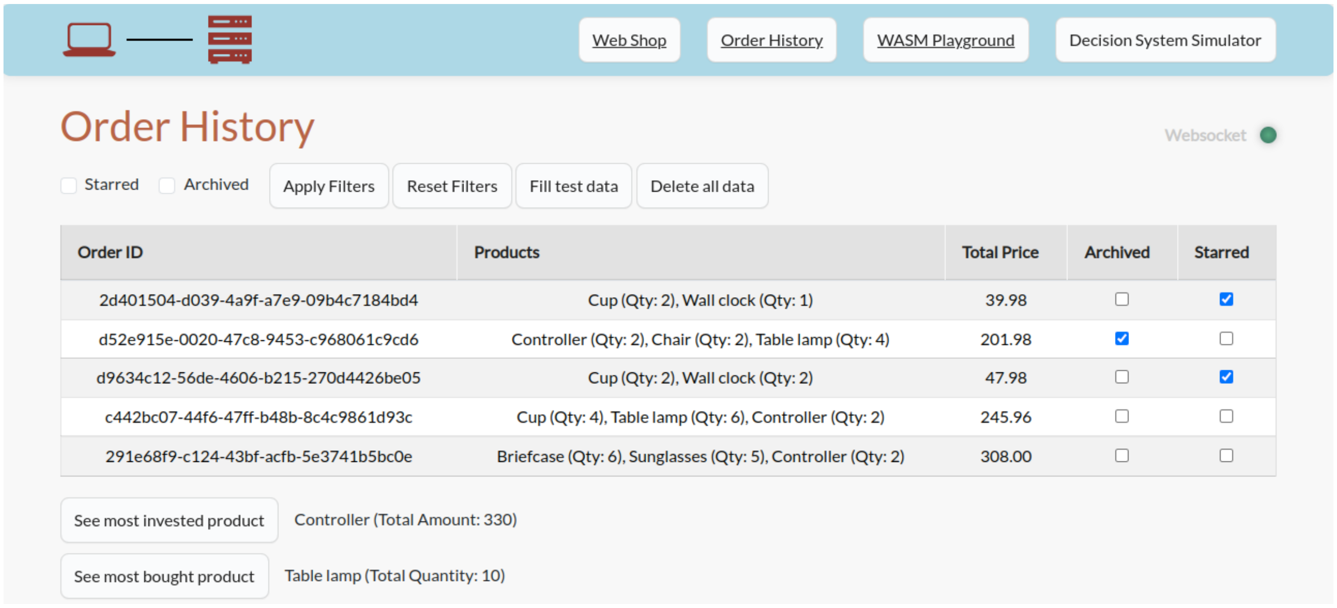


Figure 2: The demo application (Shopping page)



Order ID	Products	Total Price	Archived	Starred
2d401504-d039-4a9f-a7e9-09b4c7184bd4	Cup (Qty: 2), Wall clock (Qty: 1)	39.98	☐	☑
d52e915e-0020-47c8-9453-c968061c9cd6	Controller (Qty: 2), Chair (Qty: 2), Table lamp (Qty: 4)	201.98	☑	☐
d9634c12-56de-4606-b215-270d4426be05	Cup (Qty: 2), Wall clock (Qty: 2)	47.98	☐	☑
c442bc07-44f6-47ff-b48b-8c4c9861d93c	Cup (Qty: 4), Table lamp (Qty: 6), Controller (Qty: 2)	245.96	☐	☐
291e68f9-c124-43bf-acfb-5e3741b5bc0e	Briefcase (Qty: 6), Sunglasses (Qty: 5), Controller (Qty: 2)	308.00	☐	☐

See most invested product — Controller (Total Amount: 330)

See most bought product — Table lamp (Total Quantity: 10)

Figure 3: The demo application (Order history page)

After ordering some items, the user can then see their order history, as shown in figure above. Orders in the history can be starred or archived for later reference. The view can be filtered to show only the starred or archived orders. Below the order history table, there are two buttons labeled ‘See most invested product’ and ‘See most bought product’. When clicked, these buttons display the respective product along with relevant details like the total amount invested in that product or the total quantity bought just to its right. The most invested product is the one that the user has spent the most money on, and the most bought product is the one that is bought the most in terms of total quantity.

Task and Experiment

All participants of this evaluation are asked to perform some pre-defined tasks to get familiar with the system. The participants can access the codebase and required infrastructures in the cloud from a web browser with these details:

url: http://34.174.167.236:8080/?folder=/home/dx/project/dx/web-application{participant-id}

password: 26434b954bc74c77c11aa545

Shown in figure 3, the functions to find the ‘most invested product’ and ‘most bought product’ are currently implemented in JavaScript. The participants are asked to implement these functions in Rust, make them mobile and then replace their usage in JavaScript with our exported functions. The actual logic of the functions is already provided in the form of helper functions in Rust. These helper functions take reference of the input as parameters, which prevents them from being made mobile (because references are not supported as parameters for mobile functions). So, the actual task that has to be completed by the participants are explained step-by-step as below. There are three main phases, followed by a testing phase to conclude the experiment.

Step 1: Specification of mobile functions

First step is to create two functions in our Rust code and specify them for mobility.

1. Open the file: project_root/runtime-code-mobility-demo/src/shared/webshop /developer_experience_evaluation.rs.
2. At the bottom of this file, create two new functions. The functions signature will be similar to **get_most_bought_product** and **get_most_invested_product** in the same file respectively. Only difference is that these new functions must take the actual value as parameter rather than references.
3. From these functions, call their respective helper functions by supplying the reference of the parameter (with the &), and return the value.
4. Mark the two newly created functions as mobile with ***/// @mobile*** doc comment on top of the functions.

This completes the first phase. We have created two functions and specified them for mobility using the ***/// @mobile*** doc comment.

Step 2: Generating WebAssembly modules and JS glue code

Now, we need to run the wasm-generator infrastructure so that it generates WebAssembly modules and JavaScript glue code for our mobile functions.

5. Run the wasm-generator application from the web-ide’s terminal. Just type ‘wasm-generator’ and press enter. It takes roughly 3 minutes to complete.
6. When it finishes, the two functions from rust will be now available in the client side.

Step 3: Using the mobile functions

Finally, now use the mobile functions in JavaScript that we specified in Rust in step 1.

7. Open the file: project_root/runtime-code-mobility-demo/src/frontend/static/js/OrderHistory/index.js
8. In this file, replace the function invocations for **get_most_bought_product** and **get_most_invested_product** with the relevant function you created by importing them from initCodeDistributor.js.

Step 4: Testing

To test if everything works as expected,

9. Start the code distributor and the demo application.
    - ➔ Open the web ide terminal in two tabs.
    - ➔ Navigate to project_root/code-distributor in one and to project_root/runtime-code-mobility-demo in another.
    - ➔ Execute command “cargo run” in both.
    - ➔ Wait for the project to compile and the web server to start in both terminals.
10. In your browser, navigate to: http://34.174.167.236:8081. With the help of decision system, it should now be possible to change the execution location of the two actions shown in figure 3. The top left corner of the UI has a client and server icon. When a mobile function executes on either the client or the server, the proper one flashes.
11. The decision system chrome extension is available as a zip file in the codebase. Download and install it in your chrome browser. After installation select the extension and put the base url of code-distributor in the given input which is, http://34.174.167.236:8082. As a shorthand, a simple interface for the decision system is also available on the demo application itself.