TU WIEN Informatics

TODO: title deutsch

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Jannis Adamek
Matrikelnummer 11809490

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Lektor Dipl.-Ing. Dr.techn. Markus Raab

Wien, 29. April 2024

Jannis Adamek	Markus Raab

TU WIEN Informatics

TODO: title

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Jannis Adamek
Registration Number 11809490

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Lektor Dipl.-Ing. Dr.techn. Markus Raab

Vienna, 29th April, 2024

Jannis Adamek	Markus Raab

Erklärung zur Verfassung der Arbeit

Jannis Adamek

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 29. April 2024

Jannis Adamek

Acknowledgements

TODO: acknowledgments

Abstract

PermaplanT is a web application for collaborative online garden planing. TODO: write abstract

Contents

CHAPTER 1

Introduction

PermaplanT is an aspiring web application initiative that strives to enable users to plan gardening spaces following the principles of packagesermaculture. That includes designing a self-sustainable system that mimics a natural ecosystem with diverse plants. It emphasizes sustainability and minimal environmental impact over high yields.

PermaplanT is structured as a classic web application, divided into a client-side frontend executed in a Web Browser and a server side backend. The frontend uses JavaScript and the React framework, while the backend is implemented in the Rust programming language and uses the PostgreSQL relational database for data storage. Apart from the application code, PermaplanT features a database of about 10.000 plant entries with information about life cycles, fertility, light and water needs, and plant relationships.

The main feature of PermaplanT is the planning tool, where users can add elements to their maps. These elements include plantings (instances of plants), drawings (2d shapes that help visualize and label the users map), images and shaded areas. The planning tool is fully collaborative, any changes made by one user are directly reflected on all clients viewing the same map.

In this Thesis we will investigate whether the current implementation of the event broadcasting mechanism is performant enough to allow 100 simultaneous users making changes on the same map with reasonable performance. Specifically, we will examine:

- How much longer requests take to resolve for the user making the changes when there are 100 observers.
- How long it takes for the changes to reach all observers.

1.1 Methodology

To evaluate the performance of the event broadcasting mechanism in the PermaplanT planning tool, we conducted a series of performance tests simulating 100 simultaneous users. These tests aimed to measure the response time for user requests and the propagation delay of changes to all observers.

1.2 Background

When a user opens a map in the Browser, the frontend makes an HTTP get request to `/api/updates/maps`, which opens a one-directonal stream from the server to the client via Server Sent Events *SSE* (with a fallback to long-polling if SSE is not supported). On this stream the client receives different events in `JSON` format that are called Actions in the context of PermaplanT. The client updates its state about the map and that's how all clients stay in sync.

In the current implementation of PermaplanT, when the client makes an HTTP request to update the map, the request doesn't get resolved until all observers have received the update. If we have a look at the implementation of `POST /api/plantings/` which places new plantings onto the map.

```
1   #[post("")]
2   pub async fn create(
3       path: Path<i32>,
4       new_plantings: Json<ActionDtoWrapper<Vec<PlantingDto>>>,
5       app_data: Data<AppDataInner>,
6       user_info: UserInfo,
7   ) -> Result<HttpResponse> {
8       let map_id = path.into_inner();
9
10      let ActionDtoWrapper { action_id, dto } = new_plantings
11        .into_inner();
12
13      let created_plantings = plantings::create(
14        dto,
15        map_id,
16        user_info.id,
17        &app_data
18      ).await?;
19
20      app_data
21          .broadcaster
22          .broadcast(
23              map_id,
24              Action::new_create_planting_action(
25                created_plantings.clone(),
26                user_info.id,
```

```
27                     action_id
28                 ),
29             )
30             .await;
31
32         Ok(HttpResponse::Created().json(created_plantings))
33     }
```

In Rust `async` functions are compiled into a state machine, where each `.await` represented a state [SA24]. After each await the asyncronous executor Tokio, the implementation of the event loop we use, can make progress on another async function.

First we create a planting. After the request to the PostgresSQL database has returned successfully (the `?` operator bubbles up the error to the caller), the state machine advances to the broadcasting of all users. Finally, it resolves the request.

- ...
- ...

1.3 Goals

Bibliography

[SA24] Deepa Ahluwalia Sandeep Ahluwalia. Understanding async await in rust: From state machines to assembly code. https://www.eventhelix.com/rust/rust-to-assembly-async-await/, 2024. Online, Accessed: 2024-07-30.