A Super Smash Brothers Tournament Locator in F♯

Tyler P. Berkshire
Department of Computer Science
University of Dayton
Dayton, Ohio USA
berkshiret1@udayton.edu

ABSTRACT

A worldwide tournament locator for one of Nintendo's popular video game series, *Super Smash Brothers*, is implemented in the F♯ programming language. The system is deployed as a web application and utilizes the Fable F♯ compiler to convert the ecosystem into React and JavaScript elements. Our application is structured with the Elmish architecture's "model view update" style in mind and uses Webpack to bundle its modules. The tournament data is gathered from smash.gg's GraphQL API with API queries implemented in Python.

KEYWORDS

Elmish, Fable compiler, F♯ programming language, GraphQL, Python, React.

ACM Reference Format:
Tyler P. Berkshire. 2019. A Super Smash Brothers Tournament Locator in F♯. In *Proceedings of CPS 452: Emerging Programming Languages (CPS 452),* Saverio Perugini (Ed.). ACM, New York, NY, USA, 3 pages.

1 INTRODUCTION

Functional programming methodologies are often overlooked by developers when considering platforms for large-scale applications. This is understandable, as most mainstream GUI programming involves an imperative or object-oriented approach due to their rich and stable history as staples in front-end development. In addition, many challenges presented by functional application development require extremely creative solutions to solve. We refer the reader to one such solution for reactive HTML forms [1]. Problems like this are not inherent to other programming paradigms, leading developers to shy away from using functional languages in their software.

However, the support for functional programming has increased in recent years. While the functional paradigm may not be an optimal solution to every programming problem, the inherent methodology of functional programming can hold unmeasured subjective value for developers [2]. Immutability, pattern matching, and concurrency are all aspects of the functional paradigm which fit conceptually well into the user interaction and experience development space.

CPS 452, Fall 2019, University of Dayton, Dayton, Ohio 45469—2160

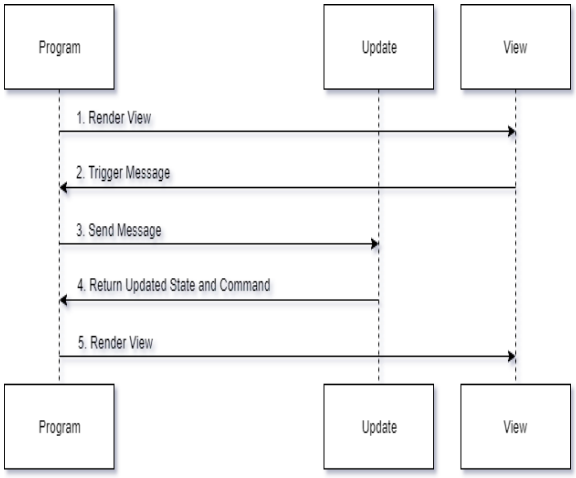

Figure 1: The main Elmish dispatch loop. Figure regenerated and adapted from https://elmish.github.io/elmish/

2 IMPLEMENTATION

Thus, we chose to develop our application in the F♯ programming language (https://fsharp.org/) in an attempt to showcase the capabilities of the functional programming paradigm on a modern web application. Our goal was to develop a user-friendly system to display the nearby, upcoming, and past competitive tournaments for one of Nintendo's popular video game series, *Super Smash Brothers*. The Fable compiler, the Elmish architecture, and React components were used to create this web application. Data for tournaments was collected using smash.gg's API with Python GraphQL queries.

2.1 The Fable Compiler

The *Fable* compiler is a powerful tool which allows F♯ to become a first-class citizen of the JavaScript environment. This means the functional programming paradigms of F♯ can be fully utilized on the web. Pattern matching, in particular, is especially useful in the application's implementation as the Elmish architecture thrives off of message and data passing. This could cause problems in more complex systems, however, as pattern matching is often troublesome with abstract data types [3]. The type safety and inference of F♯ also allows for the user interface to be structurally sound while letting the code stay concise.

Table 1: Elm concepts translated to Elmish.

Concept	Elmish Translation
`Model`	Immutable application state.
`Message`	Discriminated union representing a change of state.
`Command`	Carrier of instructions.
`Init`	Produces initial state and commands.
`Update`	Produces new state given the previous state and commands.
`View`	Renders a declaratively built UI.
`Program`	Combines the above to produce the interface.

2.2 The Elm and Elmish Architectures

Elmish is a package for Fable applications which enables the developer to utilize a "model view update" style of architecture. This style of application flow was popularized by the Elm Architecture for designing interactive programs. Elmish provides Fable with the core abstractions necessary to follow the Elm design style. This allows us to develop a solid core application that is largely UI-independent and easily reusable.

Figure 1 shows the main dispatch loop for the Elmish architecture. The page is instantiated with an initial model which is rendered by the `View` function. Once the application catches an event trigger, it sends a message through the main program to the `Update` function. This, in turn, creates a new state for the application with any additional commands and sends it back to the program for an updated render.

This structure is made possible through the Elmish abstractions of the Elm architecture. Table 1 shows how these concepts are translated into the Elmish package. The core of the application is centered around the `Model` which contains an immutable data structure representing a snapshot of the system. This model is updated whenever a new message is received in the `Update` function. The `View` method is then able to create or modify any aspects of the UI that have changed since the last update with the help of a renderer, like React. The `Program` then wraps this all together to produce a view from the model.

2.3 React Components in Fable

The `View` function depends on a renderer library to implement the actual UI. This makes the core program independent of any particular framework and provides large reusability to Elmish applications. The application can even run entirely without a UI if desired. Currently, only React and React Native have been implemented in Fable for rendering. Since our application is for the web, we used React to render our UI components.

Two of the most important features of React our application utilizes are function components and hooks. Function components are actual entities in the Document Object Model (DOM) of the application, meaning we can set triggers for different stages of a component's life cycle. In combination with React's hooks, we can completely manipulate components based on user initiated events and the current DOM state.

Listing 1 shows an example of a function component. In this example, the main application component checks the internal state of the model. If the application is currently loading tournaments, then it renders a `fallback` component. Otherwise, the application will pattern match against the CurrentRoute and load the appropriate page. This is piped into the `layout` function which defines the overall UI structure. The function component is utilized here to skip the rendering of a component if its props have not changed.

```
let App =
    FunctionComponent.Of (fun () ->
        let model = useModel()
        if model.IsLoadingTournaments then
            fallback
        else
            match model.CurrentRoute with
            | Some(Route.Root) ->
                HomePage()
            | Some(Route.Tournaments _) ->
                TournamentsPage()
            | Some(Route.Detail _) ->
                DetailPage()
            | None ->
                h1 [] [str "404"]
        |> layout
    , "App")
```

Listing 1: Function component example.

2.4 Python GraphQL Queries

The tournament data for the web application is collected through Python GraphQL queries to the smash.gg API. Listing 2 shows a simple GraphQL query.

```
result = client.execute('''
query TournamentsByLocation() {
  tournaments(query: {
    perPage: 10
    filter: {
      location: {
        distanceFrom: "30,-81",
        distance: "50mi"
      },
      past: false,
      videogameIds: [
        1, 2, 3, 4, 5
      ]
    }
  }) {
    nodes {
      id
      }}}}''')
```

Listing 2: Python GraphQL query example.

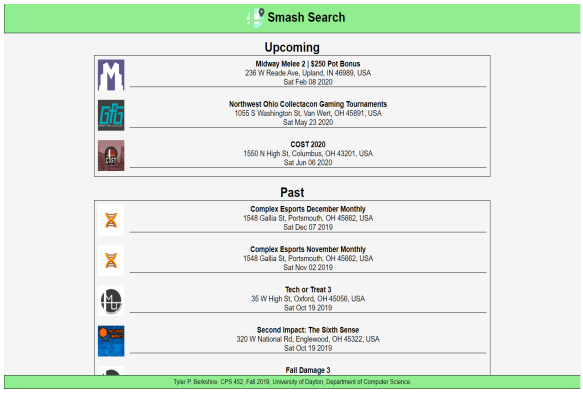

Figure 2: An example *Tournaments* page display.

This query retrieves the `id` of up to ten tournaments at a maximum distance of fifty miles from 20° latitude and -81° longitude. In the actual application, more properties of the `tournament` object are retrieved and the following fields are parameterized: `perPage`, `coordinates`, and `radius`. Once this query is received, the Python script converts the GraphQL objects to a JSON format and writes them to a file. This is done once during the application's startup.

2.5 The Smash Search Application

The web application, dubbed Smash Search, is bundled and deployed locally using the Webpack JavaScript module bundler. A Windows command-line script is used to start the application. This script runs the Python GraphQL queries to load the data into a JSON file and boots up a local server for the web application. The user can then navigate through three different page layouts on the Smash Search application. The first is the *Home* page, which only contains a button to the *Tournaments* page.

The *Tournaments* page reads the data from the JSON file, converts the objects into F♯ data types, and displays the tournament information in two lists. The top-most list displays upcoming tournaments ordered from the earliest starting date to the latest starting date. The bottom-most list displays tournaments which have already completed ordered from earliest completion date to latest completion date. Each tournament entry displays the tournament's name, venue address, and starting date. Figure 2 shows an example *Tournaments* page. Every tournament listing can be selected by the user to display more information.

Selecting a tournament will bring the user to the *Detail* page for that specific tournament. Here, more in-depth information is displayed about the tournament. A link to the smash.gg page for the tournament is also provided.

3 CONCLUSION

We presented a F♯ web application which displays the Super Smash Brothers tournament information. The Fable compiler was used to convert F♯ code into JavaScript and Webpack was used to bundle the required modules. The Smash Search application followed the Elm Architecture for interactive programs by utilizing the Elmish package. Python GraphQL queries gathered tournament data from the smash.gg API.

This project proved to be a good introduction into the functional paradigm for web development, but it lends itself to additional features. One such feature is the consolidation of the back-end and front-end processes. It would be beneficial to make the GraphQL queries from the F♯ application instead of from a separate Python script. This would enable the user to input different locations during each session rather than having to recompile every time. Another feature is the option to sort tournaments by different fields. Currently, they are only sorted by date but giving the user the capability to sort by location, number of entrants, and game type would be helpful for larger data sets.

REFERENCES

[1] J. Bjornson, A. Tayanovskyy, and A. Granicz. 2011. Composing Reactive GUIs in F Using WebSharper. In *Implementation and Application of Functional Languages*. Springer, Heidelberg, Germany, 203–216.

[2] J. D. McCaffrey and A. Bonar. 2010. A Case Study of the Factors Associated with Using the F Programming Language for Software Test Automation. In *Proceedings of the Seventh International Conference on Information Technology*. IEEE Computer Society Press, Los Alamitos, CA, 1009––1013.

[3] D. Syme, G. Neverov, and J. Margetson. 2007. Extensible pattern matching via a lightweight language extension. In *ACM SIGPLAN Notices*. ACM Press, New York, NY, 29–40.