Distributed Systems

Challenge Task - Wordle

FS2022

Olivier Lischer, Benjamin Plattner, Jan Untersander

Computer Science
University of Applied Sciences of Eastern Switzerland

Contents

List of Figures

1 Introduction

1.1 Authors

The authors of the Wordle Challenge Task are:

- Olivier Lischer
- Benjamin Plattner
- Jan Untersander

1.2 Challenge Task Requirements

The system needs to have the following components:

- Simple Frontend (e.g., HTML, Vue, React, Svelte)
- Load-balancer(e.g., traefik, nginx, HAproxy, Caddy)
- Two instances of a service (your choice), and during the challenge task presentation, one instance will be shut off.
- One storage backend

All requirements below must be met in order to pass this lecture.

- Load balancing with scalable service
- Failover of a service instance
- A websocket needs to send data from a service to the frontend. You need to implement (consume) a websocket (using e.g., Blazor that integrates websockets does not fulfil this requirement). You can produce websocket data in your backend or use 3rd party websocket services (e.g., crypto market data)
- Dockerised
- Persistent storage (storage does not need to be scalable, but you can build it scalable if you want)
- Use latest stable releases of chosen libraries and frameworks

The solution may use existing libraries and code, but those must be open source software.

You are allowed to use any language, framework, and platforms. However, the supervisors are familiar with those: Java, Golang, JavaScript, Linux.

1.3 Summary

Taking the before mentioned requirements into consideration we wanted to build a simple frontend, mainly because the focus of the Challenge Task is clearly on the distributed systems part. The recent popularity of the Wordle web application seemed to be ideally suited for such a task. It has a relatively simple user interface and it is fun to play.

For those who do not know Wordle: it is a browser-based word game where you have six attempts to guess a five letter word. For every try the correct letters are highlighted in orange. If the letter is also at the correct place it is marked green. With that, you get hints for a better next guess. There is only one word published per day, at least in its original implementation.

We intended to create a Wordle version, which is horizontally scalable and uses the Websocket protocol. The WebSocket protocol should be used for the communication between the frontend and the backend. A database should be used to store all the words and the backend should only retrieve the current word of the day from it. If user accounts are implemented it should also store user data and current game states of the users.

For the infrastructure we decided to use Kubernetes (K8s) as the container platform, MongoDB for the persistent storage and to write everything in TypeScript. As K8s comes with an in-built load-balancer, called kube-proxy, we do not need an additional one.

The local development was performed using WebStorm, where the Cloud Code plugin was used to quickly deploy development iterations to minikube, which is a local version of K8s.

2 Infrastructure

2.1 Overview

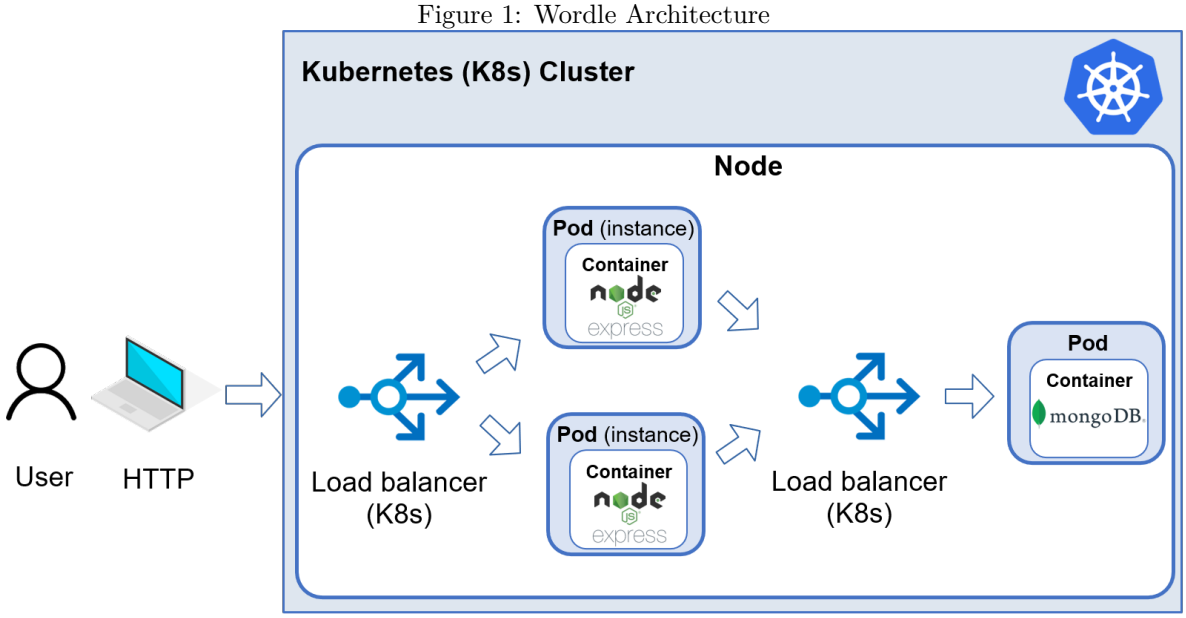
For the infrastructure we chose the following technologies:

- Cloud Code[1]
- Express[2]
- GitLab[3]
- Kubernetes (K8s)[4]
- MongoDB[5]
- Node.js[6]
- TypeScript[7]
- webpack[8]
- WebStorm[9]

The architecture section will provide an overview how the chosen technologies interact. In the Design Decision section we describe the reasons for using this particular technology stack.

2.2 Architecture

Figure 1: Wordle Architecture



The user accesses the Wordle app via browser, currently made available on localhost only. The wordle Kubernetes service load balances the traffic to a specific web server instance (a Pod) inside the K8s cluster. This Pod consists of a container that contains the Node.js express application. MongoDB is also running inside a Pod and exposed via another Kubernetes service object to the web server pods, where the current word of the day can be retrieved from the persistently stored word list.

[1] https://cloud.google.com/code/
[2] https://expressjs.com/
[3] https://gitlab.ost.ch/
[4] https://kubernetes.io/
[5] https://www.mongodb.com/
[6] https://nodejs.org/
[7] https://www.typescriptlang.org/
[8] https://webpack.js.org/
[9] https://www.jetbrains.com/webstorm/

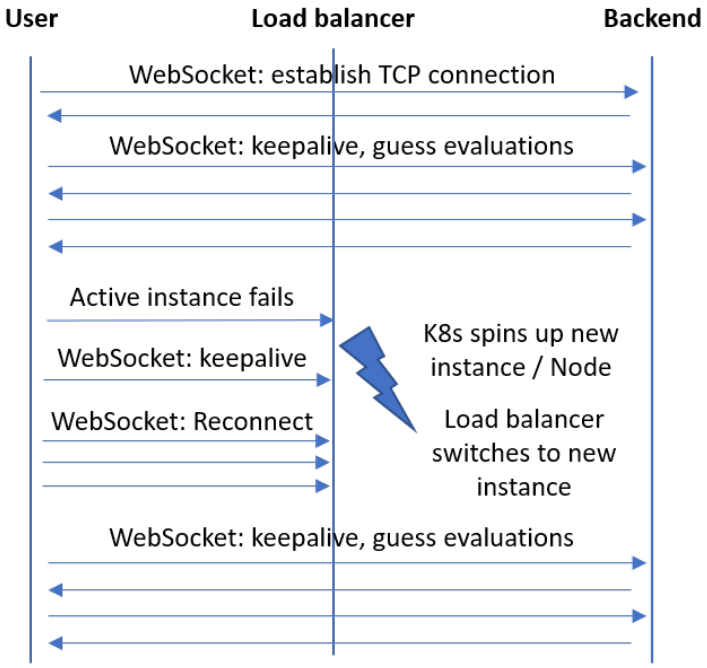
The Pods are created by a Deployment, which ensures that there are always the desired amount of Pods running. In our case the wordle Deployment ensures that there are always two web server Pods running and one MongoDB Pod. The number of Pods can be changed almost arbitrarily. If an Pod goes down, K8s spins up a new Pod. The load balancer is informed that there is again a fall-back available. The Node which contains the Pods is responsible for allocating resources (like CPU or memory).

The backend is built using TypeScript. It runs Node.js with express as the web server. WebSocket is used to communicate with the frontend, especially to send server-triggered updates. Webpack is a solution to bundle an application that has front-, backend and package dependencies into a single static asset which can be deployed.

For the collaboration and joint code development we use the GitLab free-tier versioning system which is made available by OST.

2.3 Workflow

Figure 2: Wordle Message Flow



The user loads the Wordle web app and starts playing. If a web server Pod goes down the user is notified that the app is reconnecting, as the frontend is automatically trying to reconnect. Other than that notification, there is nothing the user would notice from such a failure and fail-over.

In the backend K8s detects that the Pod is not available and the load balancer does not further send traffic to the unavailable Pod. It spins up a new Pod automatically and informs the load-balancer about it. The load-balancer switches to a running Pod, even the new one if necessary.

In the following are a few screenshots to document the application, including the demonstration of a Pod shutdown.

Figure 3: Wordle Main View and First Guess

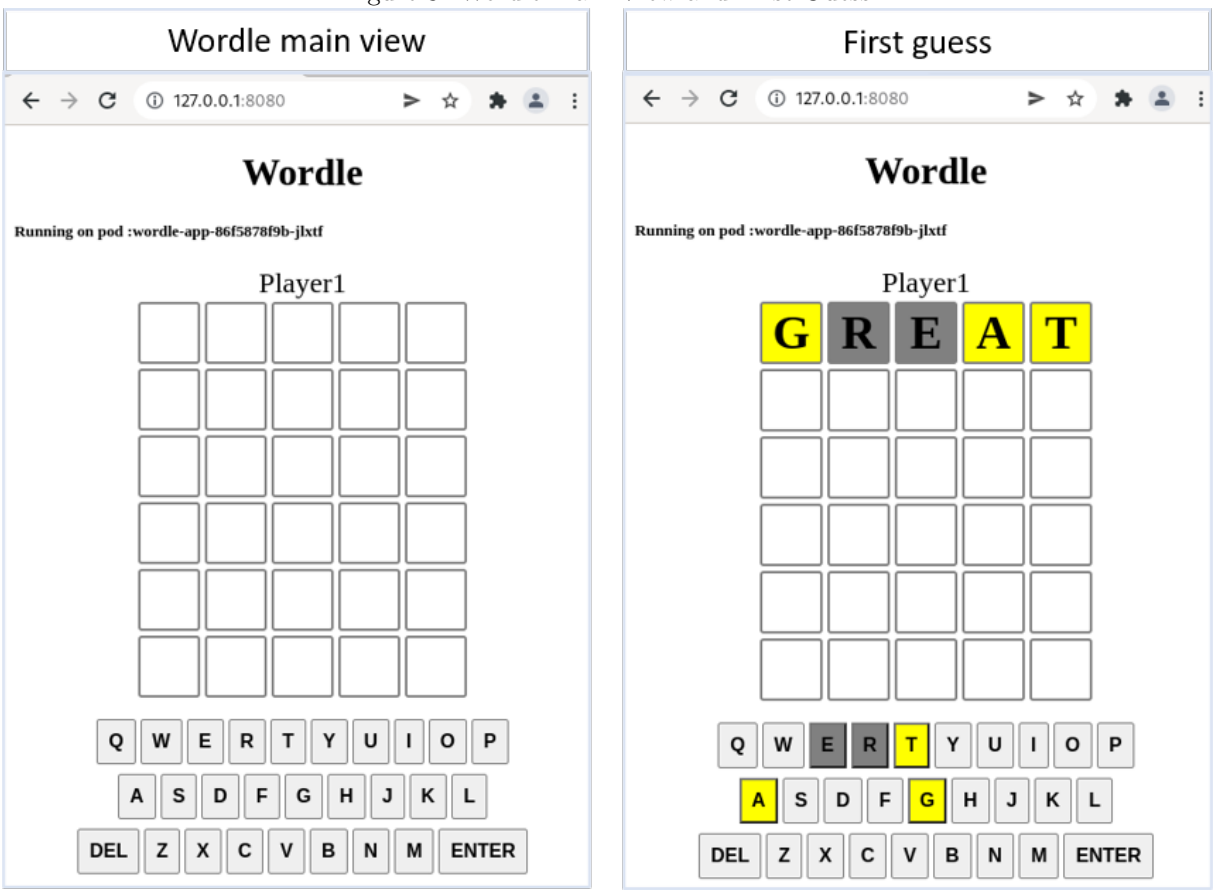


Figure 4: Wordle Guessing...

Figure 5: Instance Shutdown and Reconnection

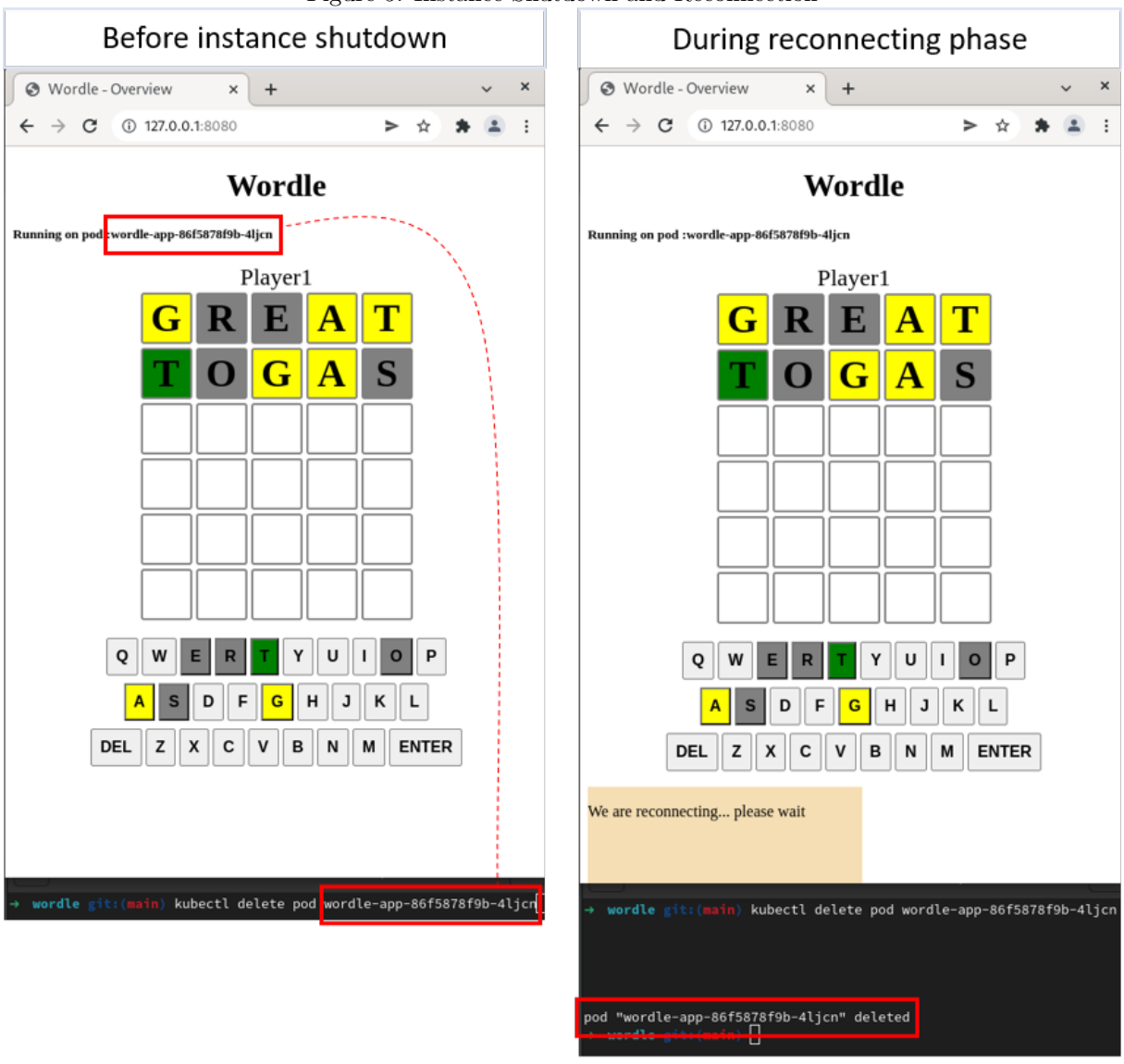
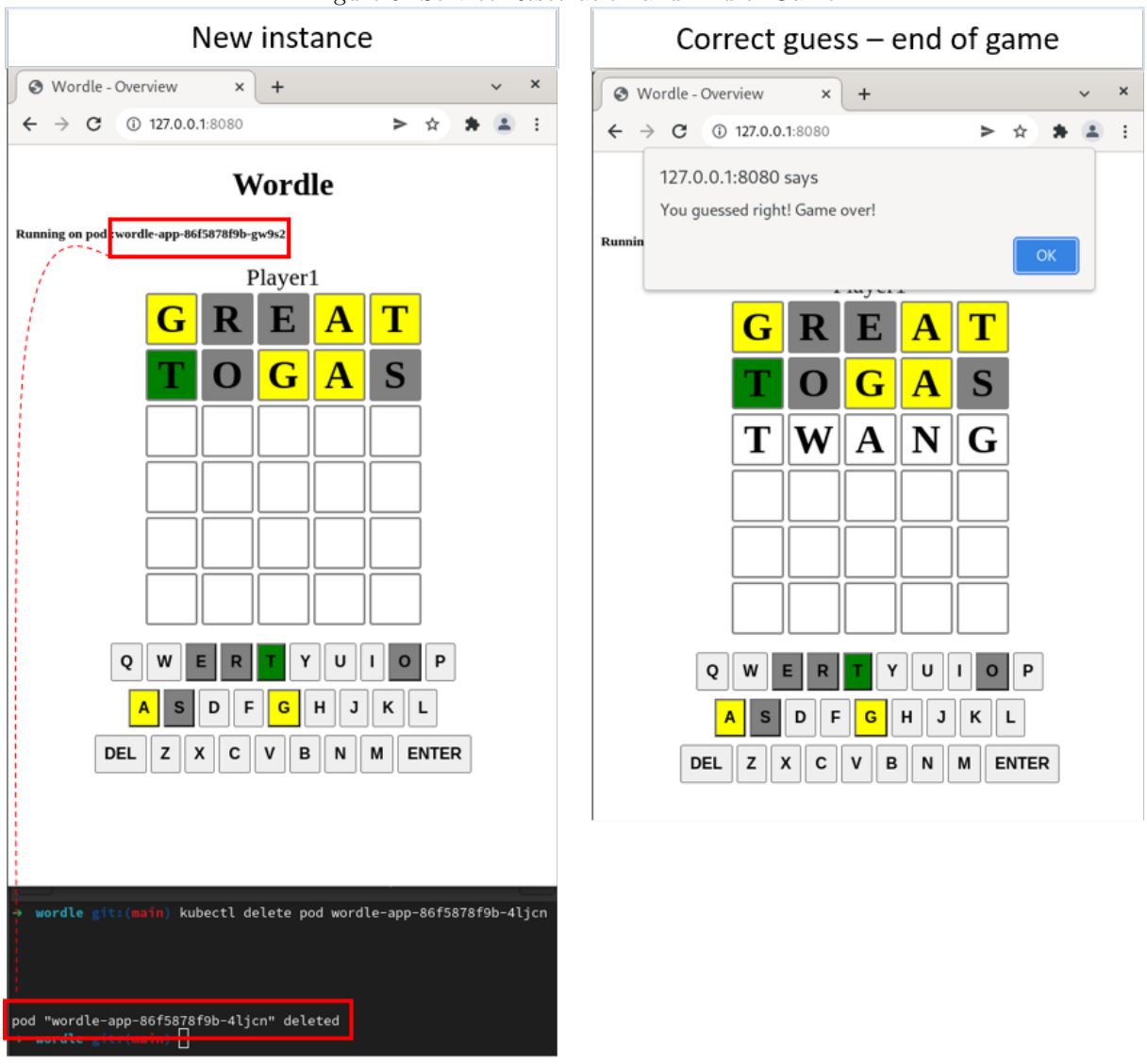


Figure 6: Service Restoration and End of Game



3 Design Decisions

A few comments on which technology we chose for a given task.

3.1 Cloud Code

When developing an application in a K8s cluster, one wants to automatically deploy new code without the hassle of manually going through the build process every time. Cloud Code is a plugin that eases this process. It is available for WebStorm and once configured, new code can be quickly deployed.

3.2 Express and Node.js

Node.js and Express are the recommended backend environments when taking web engineering modules at OST. They are widely adopted and well maintained. Also, they support Websocket, which we require for this project.

3.3 GitLab

GitLab is made available by OST as the free-tier version which is more than sufficient for our needs with this project. The main goal is to share and version code that was developed individually, but also to track progress using issues. There is no need for a more complex setup since we are a small group and the project is not overly complex.

3.4 Kubernetes

K8s allows to containerise applications. Since it also contains an in-built load balancer we quickly settled on this option. Also, we were already familiar with the technology either from the Cloud Infrastructure module offered at OST or from a recent project that was implemented using K8s.

The setup is somewhat more complicated than Docker, however, once we passed the steep learning curve, setting K8s up is relatively straight forward.

3.5 MongoDB

To store data persistently we chose MongoDB as the database technology. MongoDB is a NoSQL database which allows to store almost any object in its plain format. There are no relations between single data objects which makes it relatively easy to store e.g. JSON objects. We were also already familiar with MongoDB. Other, potentially relational, databases would be possible as well, however, for a simple storage of unstructured data the setup of relational tables would be too cumbersome.

3.6 TypeScript

Since we are software engineers who want to write clean and safe code, there is no excuse to use JavaScript any longer.

3.7 webpack

When using Node.js and basic HTML and TS it becomes difficult to deploy a web-based application that has dependencies on backend code and third-party modules. Webpack looked like an easy solution to bundle our code in a single static asset that can be run stand-alone in the frontend, but can access backend code. There are definitely better frameworks available (like React) which would do this for you, however, we are not yet familiar with React or other more modern approaches.

3.8 WebStorm

WebStorm is the recommended IDE in web engineering modules at OST. Since we are familiar with this tool and since it has many handy features like pre-installed linters, Git integration and supports TypeScript, we chose this IDE.

4 Conclusion

4.1 Learnings: Load Balancer

We noticed that the load balancer always picks the newest instance rather than a random Pod which would be the expected behaviour according to the documentation. The load balancer is managed by kube-proxy. We assume that, because the WebSocket protocol establishes the TCP connection, this behaviour is somehow not compatible with the load balancer, and the default is then to pick the newest instance. Load balancing in general works, but just not as expected. This may be a bug or an incompatibility with the WebSocket protocol.

4.2 Learnings: Webpack

When we tried to deploy our application we noticed that the interoperability between the front- and backend, as well as the imported packages does not work when using Node.js and Express. This is because the browser cannot handle imports and there is no way to forward package code from the back- to the frontend. When researching the problem we found webpack which seemed to address this issue by bundling all dependencies into a static asset. Going forward, it seems that deployments across the stack with a more modern web development framework such as React could come in handy.

4.3 Closing Remarks

The Worlde web application performs as expected. A player can access the game and play and is notified if an instance goes down and the game resumes after an automatic reconnect.

In the backend the K8s cluster handles all necessary steps to spin up instances and performs the load-balancing. The MongoDB database stores the word list and provides the current daily word to the frontend. Websocket is used to propagate notifications to the frontend that were triggered on the backend. In our case this is to send the responses of evaluated word guesses.

Overall, we are very satisfied with the outcome. The application runs stable and a user can play the game. The backend in its current implementation performs well. Potential extensions are that we upgrade the Node.js framework to a more modern one like React. Also, game progress could be stored temporarily in a cookie or localstorage so the game could be resumed after a page refresh.

We enjoyed the whole process of coming up with an idea, to think about potential implementations and then actually setting up the infrastructure. We learned a lot during the process, for example how well Kubernetes is suited for such distributed systems as it has an integrated load-balancer. Also, we noticed the limitations of simple web frameworks where the interaction between front- and backend is complicated without other third-party tools. This is a motivation for us to extend our knowledge about more modern web development frameworks.

But for now, have fun playing Wordle!