Trylinks 2023

Mantas Maciulis

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2024

Abstract

Links is a programming language developed in Edinburgh. It is a research language still in the early stages of development, so its setup for development is non-trivial, and it is hard for new Links developers to get started with it. A web application called Trylinks was developed as part of a UG4 project in 2018. Trylinks allows potential Links developers to test the language by giving users access to a shell that can run Links code snippets. Trylinks also enables users to develop actual Links web apps via a built-in code editor and a series of tutorials. The users' web apps get compiled and displayed directly within Trylinks.

Unfortunately, Trylinks is now 5 years outdated and several Links versions behind. Additionally, the deployment is currently broken, and the documentation of how it was deployed is limited, making it difficult for developers to get it up and running again or add features.

This project contributes to Trylinks by getting it redeployed and updated with the latest technological standards. Effort was invested in approaches that would reduce the risk of Trylinks becoming abandoned or unmaintained in the future, primarily by simplifying the development process. Finally, additional features and enhancements drawn from a thorough evaluation were implemented.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Mantas Maciulis*)

Acknowledgements

I would like to sincerely thank my supervisor, James Cheney, for his guidance and support throughout this project. He was always there to provide answers, and his support was invaluable to the success of this project.

Additionally, I would like to thank my mother for her support and generosity during my time at university.

Table of Contents

Chapter 1

Introduction

This chapter summarizes the contents and insights covered in Chapters 2 through 6. It aims to offer a standalone overview of the project that can be quickly and easily understood.

1.1 Motivation

Web development can be complicated, as it involves programming in several languages. For instance, creating even basic web applications typically requires using languages such as Ruby for the back-end, JavaScript for the front-end, and SQL for database interactions.

Links, a programming language developed in Edinburgh, seeks to address this issue by enabling developers to write web applications in a single, unified language, which then gets translated into appropriate code for the various layers of web development[1].

Newcomers often find it challenging to get started with Links as it is nontrivial to set up. This makes it difficult to motivate developers to incorporate Links into their projects, primarily because it is challenging for them to determine if Links aligns with their specific requirements. To solve this, a web app called Trylinks was developed as part of a UG4 project in 2018[2]. Trylinks allows potential Links developers to test the language by giving users access to a shell that can run Links code snippets. Trylinks also enables users to develop Links web apps via a built-in code editor and a series of tutorials. The user's web apps get compiled and displayed directly within Trylinks.

This project aims to build upon Trylinks 2018. The motivation stems from the project's current state. It was developed in 2018 but has not been maintained since. This means that Trylinks has now become 5 years outdated, which results in it having hundreds of vulnerabilities and many outdated dependencies. The deployment of Trylinks 2018 is broken, and the deployment details have very limited documentation, which means that it is very hard for interested developers to get it up and running again or add features.

Furthermore, a lot of additional work can be done on the platform, as highlighted by the user feedback and ideas for future work left by the previous developer [3].

1.2 Project Goals and Contributions

The project proposal allowed for a lot of flexibility in the direction of this project. As a result, the goals of this project have been identified after carefully analyzing the project proposal[4], completing a thorough evaluation of the state of Trylinks 2018, and reading previous user feedback. These goals are justified, and more detail is presented in the ”Evaluation of Previous Work” chapter.

The following goals have been identified:

- **Revitalize Trylinks:** This involves modernizing Trylinks with the latest technological standards, addressing security vulnerabilities, and redeploying it.
- **Facilitate Future Development:** This involves streamlining the development process, with a focus on minimizing the risk of the project becoming out of date again.
- **Add Incremental Improvements:** This involves implementing as many incremental improvements/features as time allows

In response to these goals, the following contributions have been made throughout this project, with a lot more detail presented in the Implementation Chapter:

Revitalize Trylinks:

- Trylinks’ vulnerabilities were removed.
- Trylinks was updated to newer technological standards.
- The Links version within Trylinks was updated to the latest version.
- Trylinks was redeployed on trylinks.net and dev.trylinks.net.

Facilitate Future Development:

- Better development strategies were adopted, with some enforced through locking branches to direct commits.
- A Continuous Integration and Continuous Delivery (CI/CD) workflow was established for Trylinks. This enables automatic deployment when developers merge their code with the production or development branches on GitHub.
- Automated cross-browser testing was implemented.

Add Incremental Improvements:

- Many enhancements were made to the code editor, notably the addition of syntax highlighting, achieved through implementing a parser.
- Security enhancements were achieved through the implementation of HTTPS.
- A new authentication system was introduced, utilizing Auth0 to delegate user management and data storage to an identity provider (idP).
- In response to user feedback, the code editor was made resizable.

- A clearer loading screen was implemented for displaying the status when users' Links programs are being compiled.
- The home page underwent a slight restyling to enhance user understanding of what Trylinks offers.
- Following the Evaluation Chapter, the stability of the Compile and Deploy pipeline responsible for deploying users' Links programs was improved.

These contributions were tested and evaluated through various techniques, including end-to-end (e2e) tests, visual inspections, and performance tests, details of which can be found in the Evaluation Chapter.

1.3 Report Outline

Chapter 2: The Background Chapter motivates the Links programming language, then uses its limitations to highlight the need for Trylinks in more detail.

Chapter 3: The Evaluation of Previous Work Chapter thoroughly evaluates Trylinks 2018. It aims to find the strengths and limitations of Trylinks 2018 to come up with ideas for direction for this project. When a limitation is found, solutions from outside of Trylinks, with a focus on open-source solutions, are investigated. The chapter concludes with a decision on what will be focused on during the project.

Chapter 4: The Implementation chapter provides a detailed overview of the project's implementation. The chapter is broken down feature by feature and described in order of implementation.

Chapter 5: The Testing and Evaluation chapter runs parallel to the Implementation Chapter and describes the testing and evaluation conducted for each feature implemented. It then conducts some general evaluation, e.g. cross-browser compatibility of the whole web app.

Chapter 6: The conclusion revisits the project's goals once again and reasons if they have been successfully achieved. It also discusses some obstacles faced throughout development and some limitations of the final application. Finally, the project's contributions are summarized, and direction for future work is given.

Chapter 2

Background

This chapter describes the process of web development and uses it to motivate the Links Programming Language. Then, it explores the limitations of Links to highlight the need for Trylinks and briefly describes what Trylinks aims to do. More detail regarding Trylinks 2018 is mentioned in the Evaluation of Previous Work chapter.

2.1 Web Development

Websites have evolved from simple static HTML documents to today's dynamic, app-like experiences[5]. This evolution has been driven by technological advancements and the growing demands of users, as well as the internet's expansion, which necessitates the creation of robust and scalable applications.

As these web applications grew in sophistication over the years, so did the complexity of their development. Developers now often require proficiency in multiple programming languages to build modern web applications. For example, knowledge of CSS, HTML and JavaScript for the front end, Node.js for the back end, and database communication languages like MongoDB may be necessary as a minimum. This modular approach is the norm because web applications are usually distributed programs, often split between the client (browser), server, and database.

Although this modular approach has its benefits, such as specialization, it also introduces challenges in web development. One of the primary difficulties in modern web development is the broad knowledge base it demands from developers in order to produce something significant. They are expected to possess expertise in multiple languages and frameworks whilst comprehending their intricacies and adhering to best practices. This can be overwhelming and time-consuming, especially for those starting their journey in web development, and may inadvertently cause these individuals to make costly mistakes or implement security vulnerabilities.

2.2 The Links Programming Language

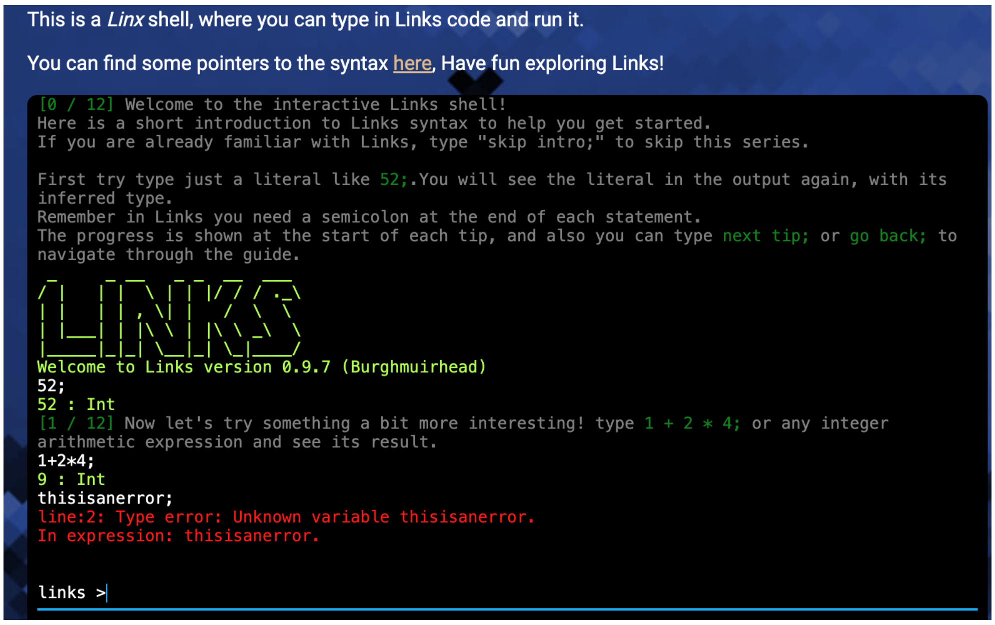
The Links programming language aims to address these challenges and simplify web programming by consolidating the distinct tiers of web development into one unified framework[1]. By integrating server-side, client-side, and database functionalities into a single language, Links provides a streamlined development process that eliminates the complexities of handling multiple languages. It also provides the ability to check type-safety once, across all tiers.

This approach helps developers by reducing the need for context switching between multiple different languages, which is a common challenge in web development. Links achieves this by offering concise syntax that abstracts away the intricacies of working with many languages and frameworks, allowing web programs to be written in a single programming language, which is then translated into languages suitable for each tier. For those interested, an example Links program is presented in the Appendix, Listing C.1.

2.3 Trylinks

Trylinks, inspired by web applications like Tryhaskell[6] and Codeacademy[7], provides an interactive playground for those who are getting started with Links. It allows newcomers to experiment with the language before committing to the lengthy setup process required.

Trylinks makes use of a REPL (Read Eval Print Loop) interactive shell, where users can type snippets of Links code and execute them.



This is a *Linx* shell, where you can type in Links code and run it.

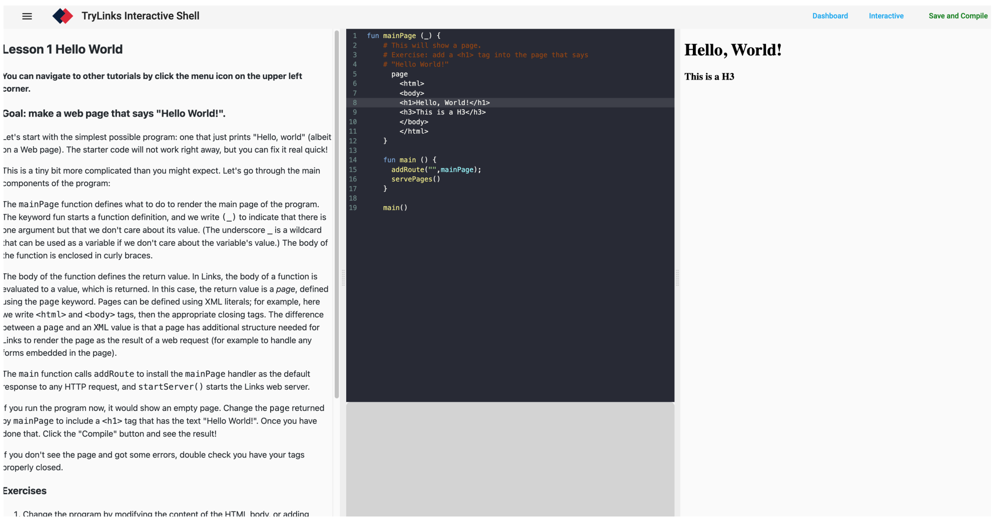
You can find some pointers to the syntax here, Have fun exploring Links!

```
[0 / 12] Welcome to the interactive Links shell!
Here is a short introduction to Links syntax to help you get started.
If you are already familiar with Links, type "skip intro;" to skip this series.

First try type just a literal like 52;.You will see the literal in the output again, with its
inferred type.
Remember in Links you need a semicolon at the end of each statement.
The progress is shown at the start of each tip, and also you can type next tip; or go back; to
navigate through the guide.

 _     _ __   _ _ __  ___
/ |   | |  \ | | |/ / / ._\
| |   | | , \| |   /  \  \
| |___| | |\ \ | |\ \ _\  \
|_____|_|_| \__|_| \_|____/
Welcome to Links version 0.9.7 (Burghmuirhead)
52;
52 : Int
[1 / 12] Now let's try something a bit more interesting! type 1 + 2 * 4; or any integer
arithmetic expression and see its result.
1+2*4;
9 : Int
thisisanerror;
line:2: Type error: Unknown variable thisisanerror.
In expression: thisisanerror.


links >
```

Figure 2.1: A screenshot of the Links REPL shell.

In addition, Trylinks aims to teach users a subset of web programming using Links. This functionality allows users to write and compile their own Links websites on the

Trylinks web app. To achieve this, an interactive tutorial series is used. This gives the user access to multiple tutorials, each with instructions and some starter code. Users are required to modify or complete this starter code, and upon execution, the code results are either displayed as a rendered page or, in case of errors, displayed in a terminal-like interface.



Lesson 1 Hello World

You can navigate to other tutorials by click the menu icon on the upper left corner.

Goal: make a web page that says "Hello World!".

Let's start with the simplest possible program: one that just prints "Hello, world" (albeit on a Web page). The starter code will not work right away, but you can fix it real quick!

This is a tiny bit more complicated than you might expect. Let's go through the main components of the program:

The `mainPage` function defines what to do to render the main page of the program. The keyword fun starts a function definition, and we write `(_)` to indicate that there is one argument but that we don't care about its value. (The underscore _ is a wildcard that can be used as a variable if we don't care about the variable's value.) The body of the function is enclosed in curly braces.

The body of the function defines the return value. In Links, the body of a function is evaluated to a value, which is returned. In this case, the return value is a *page*, defined using the `page` keyword. Pages can be defined using XML literals; for example, here we write `<html>` and `<body>` tags, then the appropriate closing tags. The difference between a `page` and an `XML` value is that a page has additional structure needed for Links to render the page as the result of a web request (for example to handle any forms embedded in the page).

The `main` function calls `addRoute` to install the `mainPage` handler as the default response to any HTTP request, and `startServer()` starts the Links web server.

If you run the program now, it would show an empty page. Change the `page` returned by `mainPage` to include a `<h1>` tag that has the text "Hello World!". Once you have done that. Click the "Compile" button and see the result!

If you don't see the page and got some errors, double check you have your tags properly closed.

Exercises

1. Change the program by modifying the content of the HTML body, or adding

```
fun mainPage (_) {
    # This will show a page.
    # Exercise: add a <h1> tag into the page that says
    # "Hello World!"
      page
        <html>
        <body>
        <h1>Hello, World!</h1>
        <h3>This is a H3</h3>
        </body>
        </html>
    }

    fun main () {
      addRoute("",mainPage);
      servePages()
    }

    main()
```

Hello, World!

This is a H3

Figure 2.2: A screenshot of the first Trylinks tutorial.

The necessity for Trylinks stems from the complex and lengthy setup processes of the Links language. Given that it's a research language, it is not primarily designed for user-friendliness. It does have installation documentation[8]. However, setup requires either setting up an outdated vagrant virtual machine (which requires a complicated upgrade to the latest Ubuntu release) or a manual install which often involves extensive tinkering. Several developers, including myself, the other UG4 student assigned to this project, and the previous UG4 Trylinks Student[9] have reported spending multiple days on the setup and that setup on Windows is even harder. Moreover, Using Links for web or database programming is non-trivial as it also requires setup and configuration of a database server[10]. Finally, Links has unique syntax, which requires time to get used to.

Given the above, it is clear that the complexity makes it difficult for newcomers to assess whether Links is suitable for their needs without first investing significant time and effort into the setup. This is compounded by the fact that Links is not a popular language and new users will have very limited support options. As a solution to these challenges, Trylinks provides a platform where users can avoid all the pitfalls above and instantly experiment with Links, leading to a smoother learning curve for new developers.

Chapter 3

Evaluation of Previous Work

This chapter provides an evaluation of Trylinks as it was in 2018 and is organized across several sections: User Logic & Authentication, Interactive Shell, Tutorials, Deployment, and the Development Process. Additionally, for each section, various similar applications are analyzed with a particular emphasis on those that are open source. This chapter aims to identify the shortcomings in Trylinks and discover areas for enhancement. The chapter concludes with a decision on areas of focus for this project, motivated by the information gained from this evaluation.

3.1 User Logic & Authentication

3.1.1 Evaluating Trylinks 2018's Approach

Trylinks 2018 incorporates fundamental user authentication functionalities, including sign-up, login, and logout. A measure of security is employed using techniques such as salting and hashing user passwords. However, an evaluation has revealed the presence of bugs and incomplete elements within the authentication process, indicating a lack of polish.

- **Security Issues:** The absence of minimum password length, complexity requirements, or checks against common password lists leaves Trylinks 2018 vulnerable to weak passwords. This significantly increases the risk of brute-force attacks [11]. This vulnerability is compounded by the lack of mechanisms to lock accounts after multiple failed login attempts or to slow down the attempts, violating recommendations in the Open Source Foundation for Application Security (OWASP) Authentication Cheat Sheet [12]. While Trylinks may not handle highly sensitive data, users' tendency to reuse passwords (a practice found in 65% of users according to Google [13]) could expose their accounts on other, more sensitive platforms.
- **Password Reset Feature:** The lack of a password reset option is a significant oversight which may hinder the user experience and permanently lock users out of Trylinks. Studies have shown that users frequently forget their passwords; a

survey by HYPR found that 76% of users required a password reset in the last 90 days [14].

- **Error Handling and User Feedback:** The current system does not provide users with clear feedback when sign-up attempts fail. This lack of specificity leaves users in the dark, unable to determine whether the issue is on their end, such as due to violating a password policy or if it's a problem with the server. According to Nielsen Norman Group, clear error messages significantly enhance the user experience by reducing frustration and confusion[15].
- **Session Management:** Session management was broken, and sessions did not persist across browser refreshes, impacting the usability of the system.

3.1.2 Evaluating Other Solutions

Replit and Codeacademy are web applications that offer interactive coding environments for various languages. They were analyzed to understand how commercial-grade applications address the challenges identified in the evaluation above.

- **Social Login:** Both web apps offer social login, which allows users to sign up using existing accounts (e.g., Google, GitHub, Apple). This has been found to streamline registration, boost conversion rates and can reduce security overhead as sensitive user credentials will not be stored on the server. However, it creates reliance on third-party providers [16], leaving users unable to log in if those providers experience downtime.
- **Password Reset:** Both web apps provide password reset functionality. This seems to be implemented using an email reset.
- **Streamlined Onboarding:** The overall registration and login process in these web apps is quite seamless, requiring only 2-3 clicks. In contrast, Trylinks 2018 requires at least 6 clicks, including a somewhat redundant login step immediately after registration. Users using social login only need one click, and there is no differentiation between sign-up and login in that case.
- **Informative Error Handling:** Error handling is notably more intuitive in commercial apps, clearly guiding users through sign-up errors.
- **Session Management:** Sessions tend to last a long time in Replit and Codeacademy. The best practices seem to be 15-30 minutes for a low-risk application according to OWASP [17], however Auth0 states that this should be balanced with User Experience (UX) to avoid losing and frustrating users[18]. Replit and Codeacademy do not follow these guidelines and have sessions that last many days.

3.2 Interactive Shell

3.2.1 Evaluating Trylinks 2018's Approach

As discussed in the Background Chapter, Trylinks 2018 incorporates a REPL shell. The shell has some security measures in place. For example, certain Links language features are disabled for security purposes. The REPL shell seems to function seamlessly. It has been tested and achieved impressive accuracy and reasonable resistance to load, as highlighted by the Testing & Evaluation chapter. However, some areas can be improved:

- **Links Version:** The Links version used in the shell is significantly outdated and several major Links versions behind.
- **Security Issues:** The Trylinks REPL shell employs the Node.js child process module and WebSocket to dynamically spawn Links child processes for interactive user sessions. Although some security measures are implemented, such as authentication and limiting users to a single Links process, there are some security vulnerabilities remaining. Links code has the possibility to interact with the system more broadly and potentially read arbitrary files from the server, depending on the permission of the Links child process. Furthermore, there is no limitation on the duration of code execution. This allows malicious users to consume excessive resources. For example, they can run infinite loops across multiple accounts.
- **User Experience:** The REPL shell in Trylinks 2018 offers an introductory series of 12 commands to guide new users in using the shell. However, this process requires the user to manually input "next tip;" to proceed to the subsequent command. It might be beneficial to explore more user-friendly alternatives.

3.2.2 Evaluating Other Solutions

In search of alternative solutions, the open-source REPL platform glot.io, which supports over 40 languages, was examined. Glot.io uses containerization (e.g., Docker [19]) to safely execute untrusted code. Containerization is a method for deploying software that packages an application's code along with it's necessary files and libraries. This allows the application to operate in a separate environment on any platform. Containerization offers:

- **Fault Tolerance:** Isolated environments prevent a single faulty container from crashing the entire system.
- **Resource Control:** Containers receive allocated resources, limiting their potential to overload the system.
- **Isolation:** Containerized code is restricted, protecting the host system from malicious interaction.

Glot.io uses an open-source library called docker-run, which provides an HTTP API for running untrusted code inside transient docker containers. For every run, a request for a new container is started and deleted. Adapting or Implementing something similar to this for Trylinks may solve some of the REPL shell issues identified.

3.3 Tutorials

3.3.1 Evaluating Trylinks 2018's Approach

Trylinks 2018 offers a series of six tutorials that aim to introduce web programming using Links. The tutorials start with basic exercises, such as printing ”hello world” on the screen, to more advanced topics like database interaction. In 2023, these tutorials could not be used, as the existing Trylink Server was down, but putting that aside, various limitations were identified:

- **Evaluation System:** Previous user feedback of the system has indicated the need to implement an evaluation system to provide guidance for users who tend to be unsure if they've correctly completed the tutorial correctly [3].
- **Code Editor:** The basic code editor can be enhanced with modern features like syntax highlighting, bracket-matching, auto-indentation, and auto-completion for a smoother development experience.
- **Window Layout:** Flexibility in window sizing and arrangement could be introduced to optimize the 3-panel workspace, especially on smaller screens. This prevents a cramped workspace as currently, the code editor only takes exactly 1/3 of the screen and cannot be resized.
- **Compilation Fragility:** The Testing Chapter has revealed that the compile and deploy pipeline, responsible for compiling a user's Links code is unstable. It fails and takes the whole server down under load. A rework of the pipeline was also recommended as future work by the 2018 project [3].
- **Open ports:** When a user compiles a program, it is deployed on a random, available port. This means if a user deploys a program, anyone can view it on trylinks.net:port.

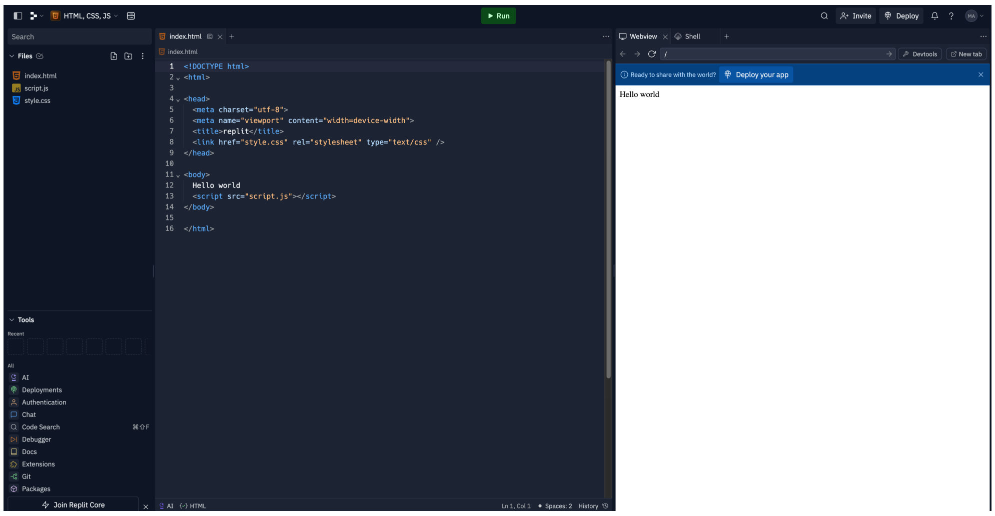
3.3.2 Evaluating Other Solutions

There are many platforms for online coding tutorials that include an IDE. Here, Replit and Judge0 are investigated for insights into how these apps handle the extra complexities web page compilation brings.

3.3.2.1 Replit

Replit offers a more robust compile-and-deploy pipeline for web development and supports simple languages like HTML to advanced frameworks like Angular. While its focus is on developing larger applications within a web browser rather than teaching programming, Replit is worth examining for its pipeline's stability.

Notable Differences:



```html
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title>replit</title>
  <link href="style.css" rel="stylesheet" type="text/css" />
</head>

<body>
  Hello world
  <script src="script.js"></script>
</body>

</html>
```

```
Hello world
```

Figure 3.1: A screenshot illustrating the Replit workspace.

- **Multi-File Development:** Unlike Trylinks, Replit allows users to work with multiple files simultaneously. This facilitates more complex web development projects where code is often separated into HTML, CSS, and JavaScript files.
- **Linux Shell Access:** Replit provides access to a Linux shell environment and the ability to execute standard commands. This strongly suggests the use of containerization for deployment, which could explain the increased stability compared to Trylinks' approach.
- **Flexible Workspace:** Replit offers a highly customizable workspace. Code windows are resizable, and users can drag and arrange them freely. This provides greater flexibility for tailoring the development environment to individual preferences.

Notable Similarities:

- **Security Vulnerability:** Despite its scale, Replit shares Trylinks' security concern: compiled programs are accessible to anyone with the link. However, Replit attempts to mitigate this by obfuscating the URL with a random subdomain (e.g. https://fcbf115f-6191-45d7-9ef6-f5fbcbafb734-00-ts999usam142.riker.Replit.dev). This likely employs a reverse proxy and database table to map subdomains to backend ports, redirecting each request to Replit.dev:port.
- **Layout:** Replit's basic layout resembles Trylinks, with both featuring side-by-side code and deployment views.

It is hard to gain much value from Replit besides ideas, as it is a closed-source solution, so an open-source solution was also investigated.

3.3.2.2 Judge0

Judge0 is an open-source online code execution system designed for flexibility and adaptability.

Notable Differences:

- **API-Driven Approach:** Users submit their code through an interface connected to the Judge0 API.
- **Queuing System:** Judge0 queues submissions, ensuring efficient handling of multiple simultaneous requests.
- **Isolated Environments:** Judge0 executes each code submission in a secure, isolated environment.

Judge0's Potential for Trylinks

Judge0 is open source and allows for secure code execution, making it a promising solution for the problems identified. The fact that Judge0 and Replit both offer some sort of containerization for code execution suggests a pattern for the state-of-the-art method for solving these problems.

3.4 Deployment

3.4.1 Evaluating Trylinks 2018's Approach

Trylinks 2018 was hosted on Digital Ocean. Despite being open-source, its deployment process was not transparent, requiring developers to handle redeployment independently or seek assistance from previous contributors when adding new features. Deploying a web app is usually complicated and requires specialized knowledge, such as server setup, network configuration, and security protocols. However, the deployment of Trylinks is even harder than this, as special users and Links databases need to be set up on the server in a specific way for the backend code to execute. Even if this project deploys Trylinks as a one off, the app will quickly become out of date again, and managing deployments for every pull request from other developers is impractical. Furthermore, my potential unavailability and tight schedule may once again cause the deployment to become abandoned, leaving subsequent developers to establish a new deployment server and domain registration from scratch or wait long periods of time for someone to redeploy the changes.

3.4.2 Evaluating Other Solutions

To address the challenges identified in the current deployment process for Trylinks 2018, alternative deployment strategies used by other applications were investigated. This research led to the discovery of Deployment Automation [20]. This method uses software tools to automate the movement of code changes through different software environments, eliminating the need for manual releases. Central to this approach is Continuous Deployment, a framework that includes automated deployment, continuous integration (CI), continuous testing (CT), and continuous feedback mechanisms. This framework ensures that code changes are automatically built, packaged, tested, and then deployed to production servers and also staging servers that mimic production environments closely but is used exclusively for testing and validation before a final release, facilitating a smooth transition to production environments. The adoption of Deployment Automation presents numerous benefits, such as increased efficiency by automating repetitive tasks that occur during deployment. It also enables faster release cycles, allowing teams to test and deliver updates more frequently. This approach aligns well with solving the inefficiencies and complexities currently present, as it will allow for deployments to be made even with my absence, and also abstract the complicated Trylinks deployment details from future developers. This suggests a promising solution to enhance the deployment process for Trylinks.

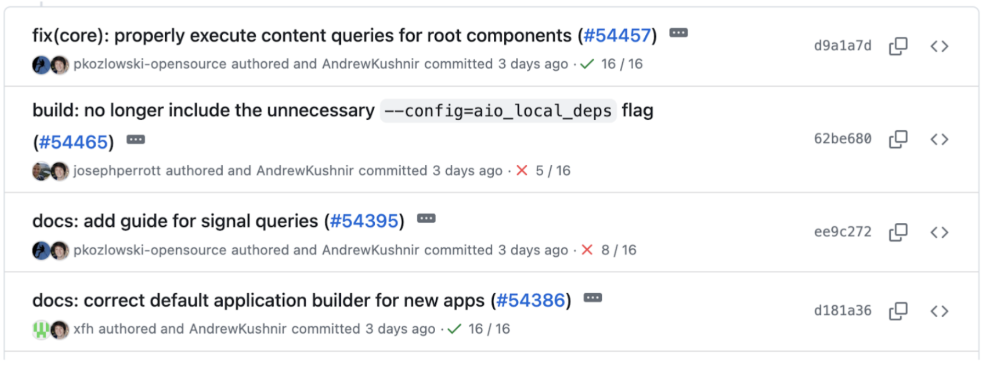
3.5 Development Process

3.5.1 Evaluating Trylinks 2018's Approach

Trylinks 2018 consisted of two repositories, one for the client [21] and one for the server [22], but the development practices within these repositories lacked several important aspects. One issue was the quality of the commit messages, which were inconsistent, lacked detail, and usually did not link back to GitHub pull requests or issues. Another area that requires improvement is the documentation. Trylinks lacked a comprehensive wiki and documentation, which makes it challenging for developers to understand the project. For example, the structure of numerous environmental configuration files was missing, creating a dependency on the previous Trylinks developer or documentation of the Links Programming Language.

3.5.2 Evaluating Other Solutions

Some other repositories were investigated to get ideas for solutions to these problems and inspiration for the points above came from the Angular Github Repo seen below, which is known for having good development practises:


fix(core): properly execute content queries for root components (#54457)
pkozlowski-opensource authored and AndrewKushnir committed 3 days ago · ✓ 16 / 16
d9a1a7d

build: no longer include the unnecessary `--config=aio_local_deps` flag (#54465)
josephperrott authored and AndrewKushnir committed 3 days ago · ✗ 5 / 16
62be680

docs: add guide for signal queries (#54395)
pkozlowski-opensource authored and AndrewKushnir committed 3 days ago · ✗ 8 / 16
ee9c272

docs: correct default application builder for new apps (#54386)
xfh authored and AndrewKushnir committed 3 days ago · ✓ 16 / 16
d181a36

Figure 3.2: A screenshot illustrating the Angular GitHub Repository.

- **Conventional Commits:** Making use of Conventional Commits[23] would greatly improve the clarity and comprehensibility of the code changes. They provide a standardized way to document changes, making it easier to track and understand the purpose and impact of each commit.

- **Better Commit Messages:** Producing better commit messages that explain the rationale behind the changes, the problem being addressed, and the solution implemented can significantly improve the development process.
- **Improved Wiki Page and Documentation:** Constructing a comprehensive wiki with information about the project's architecture, setup instructions, troubleshooting guides, and known issues would be invaluable for future developers and reduce dependency on specific individuals.
- **Use of a logging library:** The experience of getting the project up and running after a long period of time has indicated the need for detailed logging that can be viewed without access to the server. Implementing a logging library with different levels of logs (e.g., debug, info, warning, and critical) could significantly improve the troubleshooting process. This would provide a clearer picture of the system's operations and make it easier to identify and resolve issues.

3.6 The Focus of this Project

After a careful evaluation of Trylinks and consideration of the existing solutions to various issues discussed above, a direction for this project was decided before implementation began and is quoted below.

- **Revitalization of Trylinks:** The initial step will be to rejuvenate Trylinks by updating the depreciated dependencies and Links version to remove known vulnerabilities and enable users to experiment with new Links features.
- **Facilitating Future Development:** To safeguard against similar setbacks, the development process for Trylinks will be streamlined. A developer-friendly environment will be created to reduce barriers to contribution and maintenance.
- **Incremental Improvements:** After addressing the foundational concerns, focus will shift to enhancing Trylinks with additional features. Inspiration will be drawn from the points identified in this chapter and the extent of these improvements will be governed by the time available after completing the initial two objectives.

The subsequent sections of this report delve into the modifications and improvements made to Trylinks, discussed on a feature-by-feature basis.

Chapter 4

Implementation

This chapter provides a detailed overview of the project's implementation. Each feature will be presented in its own section in order of implementation. This chapter will heavily complement the Evaluation Chapter, which will run in parallel and provide qualitative and quantitative evaluation for each feature discussed here.

4.1 Getting Trylinks Running & Updated

4.1.1 Goals

Trylinks was written in Angular, a framework for building scalable web applications. For this part of the implementation, the goal was to update Angular to a recent release and update all of the packages until no known vulnerabilities remain. Finally, the Links version used for the tutorials and interactive shell should be updated to the latest available version.

4.1.2 Justification

As a minimum requirement for this project to succeed, it is clear to see that Trylinks should be able to run for the foreseeable future. Five years without development has left Trylinks 2018, including its deployment, broken and unusable due to the lack of maintenance since the website's release. All of the dependencies of Trylinks 2018 have become significantly outdated (over 150 vulnerabilities between the client and server, with 20 being critical according to the npm auditing tool). Furthermore, the Angular version used for Trylinks was Angular 7, whereas now the latest is Angular 17. Updating to newer technological standards would solve these vulnerabilities and enhance compatibility with newer packages and technologies, which could improve the performance of the application. Furthermore, using the latest and most widely adopted technologies for development would hopefully attract more developers to Trylinks and make their job easier which would facilitate future growth.

Finally, updating to the latest version of Links would allow users to learn newer Links features, which is important as teaching Links is the goal of the website.

4.1.3 Implementation

Lacking access to the server used for deploying Trylinks posed significant challenges, making it infeasible to run the program without encountering issues, even when attempting to use the same dependencies as the original web application. This was compounded by the fact that Trylinks needed a very specialized setup to run, including specific database setups and environmental configuration files that were not well documented.

Therefore, the initial step was to do a large amount of research, as well as trial and error, and gain an understanding of Links in order to reverse engineer the configuration files and environmental setup of Trylinks 2018. This also involved the setup of numerous databases required by the backend.

Following this, the ”npm update” command was executed in order to update as many of the dependencies as npm could handle. Unfortunately, it was not so simple and there were over 100 vulnerabilities remaining. However, at this stage, the web app could be built successfully, and the tutorial section of the web page seemed to work sometimes.

It would be ideal if ”npm update” could have brought everything up to date, however, the reality of managing dependencies in a software project is far more complex. Dependencies within a project are interwoven, much like a web, where each component relies on others, creating a tightly-knit ecosystem. This interconnectedness means that updating one dependency can have a domino effect on others, particularly if a critical dependency remains outdated and is a cornerstone for several others within the project.

This is an example of a repeating type of issue that occurred whilst getting everything up to date: Initially, the socket.io library on the back-end was updated, which, in turn, necessitated an upgrade to a newer version of WebSocket to maintain compatibility with the front-end. However, this updated WebSocket version was incompatible with our project's outdated Angular framework. Upgrading Angular to resolve this issue introduced further complications: ten packages that were integral to our project with the older Angular version were incompatible with the newer version of Angular. Addressing these incompatibilities would not only require updating these packages but also dealing with the fact that some packages lacked equivalents in the newer Angular ecosystem. This necessitated finding alternative packages and implementing significant changes to the code base to accommodate these new dependencies. This is alongside the changes that need to be made manually for each major Angular version upgrade and compounded due to the version of TypeScript changing.

In the end, there was no shortcut to this, and Angular had to be transitioned from versions 7 to 16 one at a time. The official Angular update guide [24] was used to ensure the resolution of all the incompatibilities caused by upgrading and lower the chance of missing a step.

After this process was completed and some code changed to match the newer ways of doing things in Angular 16, the application could be built with no vulnerabilities. However, broken aspects of Trylinks 2018 still remained.

Following this, the Links version used in the interactive shell was updated. This was trivially achieved by updating the Links version on the server. At this point, the goals for this feature were completed. We had a functioning website using the newest version of Angular, where the interactive shell and tutorial pages have been updated to work with the newest version of Links.

4.2 Redeployment & Improved Development Process

4.2.1 Goals

The following goals were identified for this feature.

1. The implementation of a development strategy that maintains a constantly deployed Minimum Viable Product (MVP) at all times.
2. The implementation of a smoother maintenance, development, testing, and deployment process for incoming developers.
3. The implementation of strategies for enhanced knowledge sharing to ensure the project's progression is independent of my future availability.

4.2.2 Justification

The 2018 Trylinks project was difficult to get started with. This was partly due to its restricted deployment setup and development process, as outlined in sections 3.4 and 3.5. With the previous feature completed, although not perfect and still containing bugs, Trylinks could technically be used. Therefore, it was decided that the next step should address these issues.

Some of the motivation for immediate deployment was to have a Minimum Viable Product (MVP) always deployed and ready to go. This strategy aimed to mitigate the risk of project disruption by always ensuring a functioning output. This approach also works towards achieving the second goal, involving easier deployment for future developers, which will be achieved through a CI/CD pipeline discussed further below.

The choice to improve the development process aimed to ensure the continued progress and growth of Trylinks. Prioritizing these improvements before introducing any new features seemed logical. It would benefit this project and also allow future development to serve as a measure to assess whether development had indeed become more streamlined with the introduced changes.

4.2.3 Implementation

4.2.3.1 Development Process

The first course of action was to establish development guidelines. Aiming for a modern and professional methodology, conventional commits and a branching strategy were adopted.

Conventional Commits:

Commit messages are a straightforward but effective convention that promotes uniformity in commit messages, enhancing their clarity and comprehensibility. [25]. This convention has various benefits, such as communicating the nature of changes to other developers and making it easier for others to contribute to the project by providing a more organized commit history [25].

The format is structured as follows[25]:

```
<type>[optional scope]: <description>
[optional body]
[optional footer(s)]
```

Where type indicates the kind of change (e.g., fix, feat, chore) and scope provides additional context, such as the component affected by the change.

Here is an example[26]:

```
feat(lang): add Polish language
```

Branching Strategy:

Transitioning from a single-branch development model, a branching strategy was adopted to support simultaneous development activities for this project. In this strategy, creating a branch effectively duplicates the codebase at a specific moment, facilitating isolated development that does not interfere with other branches or the main codebase. The approach involved two primary branches: the development branch and the production branch, both of which are locked to direct commits. To introduce changes, developers are required to create feature branches, following naming conventions like `feat/add-new-feature` or `fix/resolve-this-issue`. These branches allow for commits and subsequent merging into the develop branch for testing, before a final merge into the production branch. The rationale behind maintaining separate production and development branches becomes apparent in the context of the web app's deployment strategy, discussed further below.

Adopting a feature-branch based workflow offers significant benefits for team collaboration as it allows developers to work on different features at the same time. However, even for solo developers, this approach is valuable as it provides isolated development environments and ensures that ongoing work does not disrupt the main branch.

When a feature branch reaches a point suitable for testing or deployment, we employ a squashing and merging technique to consolidate changes into the development branch. This condenses the multiple commits for that feature into one commit describing the feature. This method keeps a clean and concise commit history on the primary branches. However, this does not result in a loss of detail, as the commit history can still be accessed via the linked pull request. For those who prefer a visual diagram of this process, it is provided in the Appendix, Figure C.1. Parts of it will become clearer after the next section.

4.2.3.2 CI/CD Pipeline

To ensure that the goals (1) and (2) were met, a CI/CD (Continuous Integration and Continuous Delivery) pipeline was implemented.

Continuous Integration (CI): Under this framework, developers work independently on their code segments. Upon completion, they initiate a merge request to the central repository, triggering the CI process. During CI, a suite of automated tests written by the developers is executed to verify the code's integrity and to prevent the introduction of known bugs into the branch. The application also undergoes a build process to guarantee that only buildable code is merged into the main repository.

Continuous Delivery (CD): Successful CI transitions into the CD phase, focusing on application deployment. The process typically incorporates multiple deployment stages, including 'develop' for internal use by developers and 'production' for the end users, facilitating a clear separation between development and production environments. Develop may be hosted on dev.webapp.com, whereas production could be hosted on webapp.com.

This CICD process offers significant advantages for Trylinks and aligns well with the goals for this project. Complicated deployment details will be abstracted away from future developers. Furthermore, their workflow will be streamlined: they'll check out feature branches, implement their changes, and create pull requests for merging into the 'develop' branch. Once an admin approves the merge, the developer can immediately test their work on the live web app. Developers can also view logs and troubleshoot build failures without requiring access to the deployment server. Finally, this approach will aid in implementing all future features and save time, as all of the deployment will be done automatically each time a feature is completed.

Some Prerequisites:

Some prerequisites had to be accomplished before implementing the CI/CD process.

The domain trylinks.net had to be registered through a domain provider to ensure users are directed to the Trylinks web application upon visiting the domain. DNS records were configured, including a 'dev' subdomain, to facilitate separate development and production environments. More benefits of the two environments are discussed further below.

A server needed to be provisioned to host both the front-end and back-end components of Trylinks. DigitalOcean [27] was selected for its affordability and reputation. A Linux-based environment was chosen, influenced by familiarity and compatibility with Links. Secure Shell (SSH) access was also configured, alongside a Links user who is responsible for all Trylinks related commands. This user was configured with restricted permissions, enhancing the security of allowing the execution of arbitrary Links code.

Nginx [28] was implemented as the web server and configured to redirect root path requests to the front end and API requests to the back end.

Implementation Details:

GitHub Actions were utilized to automate the CI/CD pipeline. A self-hosted GitHub

Actions runner was set up on the DigitalOcean server, enabling configured commands to be triggered on the server by events on the repository.

To utilize GitHub actions, a YAML workflow file had to be configured. This workflow file lists the commands to be executed on the server in response to specified repository events. The CI/CD workflow created includes dual deployment configurations for development and production branches. Development deployments are hosted on dev.trylinks.net and are initiated by pushes to the 'develop' branch, offering a live testing environment that mirrors the production setup. This allows developers to conduct thorough, real-world testing beyond their local environments. Production deployments are triggered by pushes into the 'main' branch, subject to approval by designated maintainers, ensuring that only validated changes are released. The production environment is hosted on trylinks.net.

The completed workflow configurations execute the following series of operations to automate the deployment process:

1. Securely transmit GitHub Environmental Secrets & Variables to the server.
2. Clone the repository onto the server to ensure the latest version of the code is always used.
3. Utilize Node Version Manager (NVM) to select a specific Node.js version.
4. Install dependencies for the client-side application to prepare for building and testing.
5. Build the client-side application to compile it into a deployable format.
6. Install dependencies required by the server to set the stage for back-end functionality.
7. Stop the current instance of the PM2 process manager.
8. Start the PM2 process anew with the updated application.
9. Restart the Nginx web server to apply any new configuration changes.

As mentioned above, environmental variables are securely handled through GitHub Secrets, ensuring sensitive data is safely injected into the deployment process. Apart from the security benefits, GitHub Secrets/Variables were chosen to accommodate goals (1) and (3) for this feature. This is because GitHub Secrets allow anyone with the required permissions to add, remove, or change environmental secrets/variables from the GitHub Dashboard without needing access to the server. When this was implemented for the client, some extra difficulty arose and a script needed to be created that substitutes placeholders in the Angular environment files with the actual values from GitHub Secrets right before the build process.

PM2 is a process manager for Node.js and was employed to enhance deployment reliability. PM2 ensures that the server remains operational continuously and is capable of self-recovery from failures, including unexpected crashes or system reboots.

4.3 Improving the Code Editor

4.3.1 Goals

The goal of this feature was to implement syntax highlighting and, if time allowed, additional code-editor features. Furthermore, the aim was to design this editor in a modular and expandable manner, facilitating the future integration of more code-editor functionalities.

4.3.2 Justification

The Trylinks 2018 code editor lacked useful features found in competitors' editors, such as syntax highlighting, auto-completion, and auto-indentation. They are used by the majority of developers, with 70+% of developers using a standard IDE in their workflow, which inherently offers these features [29]. One of these features, syntax highlighting, has been found to significantly reduce the time taken for a programmer to internalize the semantics of a program [30]. This implies that there is a higher mental burden to programming and understanding what is going on without syntax highlighting. Given the unique syntax of Links and it being a new language, there is already a mental burden that new developers to Links will face, and this should not be increased. In addition, if some of these features were part of Trylinks, it would help bridge the gap between a dedicated Links setup and the Trylinks web app. This would help further the goal of creating a great learning platform for new Links developers.

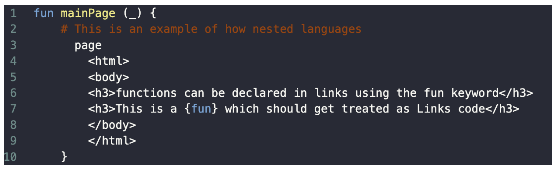
4.3.3 Implementation

Initially, tokenizer systems using regular expressions were explored to implement syntax highlighting. This approach segments code into distinct parts, assigning categories such as keywords and strings to each part. For example, a regex pattern can be utilized to identify Python keywords:

```
pattern = r'\b(if|else|for|while|def|class|print|and|or|not)\b'
```

Assigning unique colors to each category seemed straightforward with this method. However, it quickly became apparent that this approach was unsuitable for Links due to several significant limitations. The primary shortfall is a lack of semantic understanding. This leads to inaccuracies, especially in languages with nested structures like XML or Links, or in files combining multiple languages, such as JSX. This method also obstructed the goal of extensibility, as semantic understanding is crucial for implementing features like auto-completion and auto-indentation.

Adopting a lexing and parsing approach presented a more effective method. Lexing involves decomposing the code into fundamental units known as tokens. This process facilitates the identification of code elements. Following lexing, parsing analyzes the tokens' relationships based on the language's grammar, constructing a structured interpretation of the code's meaning. This method enables more precise and context-sensitive syntax highlighting by differentiating between similar elements in various grammatical roles. This also paves the way for additional features such as code navigation, basic error checking, and code completion. For example, in the following snippet of Links, differentiating between the keyword def, and the string ”def” in the HTML element is difficult with regex and may require complicated rules for edge cases.



```
fun mainPage (_) {
    # This is an example of how nested languages
      page
        <html>
        <body>
        <h3>functions can be declared in links using the fun keyword</h3>
        <h3>This is a {fun} which should get treated as Links code</h3>
        </body>
        </html>
    }
```

Figure 4.1: A simple Links program demonstrating nested structures.

In contrast, a parsing strategy understands the relationships and hierarchies within the code and can more reliably handle these types of issues.

We hoped that leveraging the existing Links lexer and parser, developed in OCaml, and employing js-of-OCaml to execute the code in a web browser could be a straightforward solution. This would involve assigning appropriate colors to each token type identified. However, in the context of a code editor, multiple challenges arose[31]:

1. The document is constantly changing, which requires syntax highlighting and parsing to adapt in real time.
2. The utilization of an expensive algorithm is impractical. If the parsing takes too long, it will introduce latency that makes editing feel unresponsive.
3. Inputs to the editor are often not in a finished, syntactically correct form. Our syntax highlighter will still have to make sense of this code, as we do not want features of the editor (e.g. highlighting) to shut down when there’s a syntax error in the document.

Therefore, this approach was deemed infeasible. This is because parsers are generally more resource-intensive than basic lexers or regex-based highlighters. Furthermore, the existing implementation is not designed for error recovery and support for changing documents. Instead, it’s designed with a focus on accuracy and parsing complete programs as part of the compilation process.

Given these complications, alternative solutions that balance complexity, feasibility, and extensibility were explored.

The next approach discovered and also adopted was the tree-sitter algorithm [32]. The tree-sitter algorithm is an incremental parsing system designed to address parsing efficiency issues. It constructs a Concrete Syntax Tree (CST) that includes all code elements, even errors and whitespace, allowing for targeted edits without full re-parsing. Error nodes are intelligently inserted to ensure continued parsing beyond the errors. Tree-sitter is language-agnostic which means that it generates parsers from specific grammar definitions. This eliminates the need for bespoke parsers for each programming language, which simplifies the implementation.

The next step involved determining how to integrate tree-sitter into Trylinks. A challenge arose as tree-sitter was written in C, which is difficult to execute in a browser environment[31]. Additionally, the aim extends beyond incorporating syntax highlighting; there is a desire for further enhancements to enhance the overall functionality and user experience within Trylinks. Consequently, a decision was made to find a JavaScript-based implementation of tree-sitter.

Subsequent research led to the discovery of Lezer, which is heavily inspired by tree-sitter and redeveloped to operate in JavaScript. This redevelopment focused on achieving a more compact framework, tailored specifically for web application use cases[31]. Following this discovery, CodeMirror 6 was identified as a compatible code editor that supports Lezer grammars.

To utilize Lezer, a developer must first write a declarative grammar file for the language they wish to implement. Once this grammar file is in place, Lezer can automatically generate a parser for that language. This generated parser can then be used as the foundation for developing various language-specific features and extensions, such as syntax highlighting, automatic indentation, and code auto-completion.

The process of integrating the Lezer grammar proved to be a substantial component of this project. It required in-depth analysis and translation of the Links tokenization and parsing mechanisms from OCaml to a compatible format for the Lezer grammar. The main challenge arose from the sheer volume of the codebase, spanning over a thousand lines, and the inherent differences between OCaml and JavaScript. Working within the constraints of a library further complicated the task, as certain OCaml features were difficult to represent in the limited grammar file syntax. To navigate these complexities, a simplified version of the Links grammar was implemented. Consequently, the grammar developed for Lezer wasn't a direct match to its OCaml counterpart. However, care was taken to ensure that all of the features needed for tutorials are supported. Gradually the foundation was expanded, but there is still more work to be done, which is detailed in the Evaluation chapter.

Following the creation of the grammar and the generation of the parser for Links, it was also necessary to develop a highlighting extension to meet the objective. This required carefully picking tags for elements of the grammar, e.g. IDENTIFIER may refer to var, fun, def. It was then necessary to decide on the colors for each tag and ensure they were visually appealing.

Integrating the Lezer Grammar into Trylinks was also quite challenging, primarily because Trylinks was built on CodeMirror 5, which lacks Lezer support. Additionally, the Trylinks platform is developed with Angular, and there wasn't an existing CodeMirror 6 component compatible with Angular that met all the desired features. Given this, It was necessary to learn how to implement JavaScript components into Angular which required new knowledge of change detection and dependency injection. GitHub was a vital tool in this, and code was adapted from this application[33], which itself is inspired by this[34], with some additions and changes to support Lezer Functionality.

Having completed this feature, some extra extensions were trivially added due to the care taken in choosing the implementation for this feature. These enhancements include a History feature to facilitate undo actions, automatic bracket closing, active line highlighting, and numerous other beneficial extensions.

4.4 Improving Security

4.4.1 Goals

The primary goal was the implementation of HTTPS (Hyper Text Transfer Protocol Secure).

4.4.2 Justification

HTTPS was chosen in order to protect user credentials in transit. Furthermore, given sufficient time, the intention was to overhaul Trylinks' authentication process through implementing OAuth-based authentication. HTTPS is a prerequisite to this as most OAuth providers require it.

4.4.3 Implementation

HTTPS, which stands for Hypertext Transfer Protocol Secure, is an internet communication protocol that protects the integrity and confidentiality of data between a user's computer and the server. Websites transition from HTTP to HTTPS by acquiring a Secure Sockets Layer (SSL) certificate from a trusted Certificate Authority (CA). When a web browser attempts to establish a connection via HTTPS, it verifies that the SSL certificate corresponds to the domain name being accessed through a process known as the SSL/TLS handshake. This certificate secures data by encrypting it using its public key, making it unreadable while it's being transmitted. The owner of the domain possesses a private key, which decrypts the data upon arrival at the server[35].

Certbot was installed on the Trylinks server to automate the acquisition and installation of SSL/TLS certificates from Let's Encrypt. This process involved obtaining a certificate for the domain and configuring Nginx to utilize this certificate, redirecting all HTTP traffic to HTTPS. Furthermore, an automated renewal job was established to refresh the certificate at regular intervals.

Despite the successful implementation, testing revealed a significant challenge. User's Links programs are compiled on random, available ports in the back-end and then accessed in an iFrame, next to the code editor as http://trylinks:port. Switching to HTTPS caused the inability to display users' compiled web applications within the iFrame. This issue arose because modern browsers' security settings prevent the mixing of HTTP and HTTPS content. Initially, it appeared simple to just use HTTPS for the range ports used for Trylinks programs, but there were complications.

SSL/TLS certificates are bound to a domain name and are configured to work on standard ports (443 for HTTPS) in web server configurations like Nginx. When https://trylinks.net is accessed, the SSL certificate is applied because it matches the domain name and uses the standard HTTPS port. Then in the Nginx configuration, we can forward paths to different internal ports. For example, the path / gets redirected to the running angular app, whereas /api to port 4000, where the server is running.

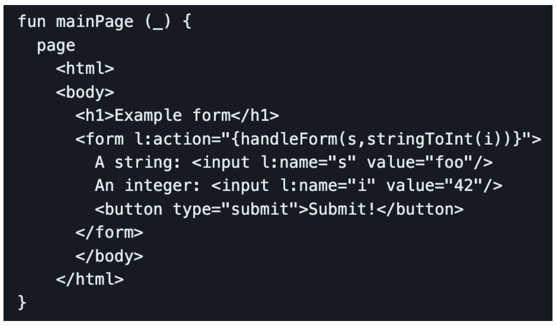
The problem is that this does not work with arbitrary ports because this needs to be configured in advance. It's not feasible or efficient to set up these configurations for thousands of ports that are required for an application such as Trylinks. Even if there is a workaround, this is not a good design.

Instead, an obvious solution would be to make the port number part of the path and then use a reverse proxy to route to the internal port based on the path. For example, trylinks.net/port/[NUM] could internally be routed to localhost:port. This was implemented, but a few issues arose:

1. Nginx automatically appends the path after routing to the port on the server, so instead of routing to localhost:port, Nginx routed to localhost:port/port/NUM. This can be circumvented using regex rules but their power is limited in Nginx, and certain behaviours and edgecases still caused issues.

2. This violated our extensibility goal. If developers change the path in the future, the tutorial functionality will break as the regex will no longer work. This will be a hard bug to identify without access to the Nginx configuration.

3. Links processes sometimes need to add arguments to the path (see Links code below). This further complicates the regex, as now we need to correctly rewrite /port/3003/url/lname=xxx;sname=yyy to localhost:port/lname=42;sname=foo. Nginx is too limited to reliably handle this in all cases.



```
fun mainPage (_) {
  page
    <html>
    <body>
      <h1>Example form</h1>
      <form l:action="{handleForm(s,stringToInt(i))}">
        A string: <input l:name="s" value="foo"/>
        An integer: <input l:name="i" value="42"/>
        <button type="submit">Submit!</button>
      </form>
      </body>
    </html>
}
```

Figure 4.2: A simple Links program with arguments.

The final solution identified and used was to register a wildcard domain with our domain provider. This means that anything.trylinks.net is valid and will be sent to our server. Then, SSH was configured for this wildcard domain, and Nginx was set up in order to forward to any port based on the subdomain as long as it is a number. This means that https://4844.trylinks.net will now show the user's compiled program running on port 4844. Since nothing is added to the path, all the complications discussed are eliminated.

4.5 Improving Authentication

4.5.1 Goals

The goals were to implement a new authentication system with the following functionality: social login, longer sessions, no password storage on the server, password reset, password lockout, and password complexity requirements.

4.5.2 Justification

As discussed in the Evaluation Chapter, user authentication was quite lacking in Trylinks. Additionally, the fact that Trylinks allows for the execution of Links code on the server, which has the potential to interact with the rest of the system may allow a malicious user to expose sensitive user information stored on the server, such as password hashes and emails.

4.5.3 Implementation

The best solution identified to mitigate these risks was to ensure that Trylinks never directly handles emails and passwords. Therefore, the implementation for this feature headed in the direction of OAuth-based authentication with a trusted Identity Provider (IdP).

IdP-based authentication utilizes a third-party service to verify user identities, providing a secure login mechanism that eliminates the need for web apps to manage usernames and passwords directly. Security is improved as it is offloaded to specialized providers.

Auth0 was chosen as a provider for Trylinks. Auth0 stands out with its generous free plan, capable of supporting up to 7,500 active users [36]. Furthermore, it supports all of the features found lacking in Trylinks. Finally, Auth0 complies with the General Data Protection Regulation (GDPR) and many more data and privacy compliance standards which can be found here[37].

Front-end Implementation

Implementing Auth0 on the front end was relatively straightforward. Initially, configurations were made on Auth0's website to set up security settings e.g. allowed callback URLs and to obtain essential credentials (Domain, ClientID, Client Secret), which were then securely stored and dispatched via CI/CD. Following this, the Auth0 Angular SDK was installed, and the authentication setup was done according to the official guide here[38]. This included creating components and configuring callbacks. Route handling was previously implemented by redirecting the user to the login screen if the "username" cookie is unknown. This has been enhanced with route guards that authenticate user access to certain paths using Auth0. Finally, the login experience was customized to include all of the features needed to meet the goals and to match the platform's branding. The final login page can be seen in the appendix, Figure B.3.

After a user authenticates using this process, they get routed back to Trylinks and are granted a JSON Web Token (JWT) token, which was configured to be included in every subsequent request to the Trylinks server. Refresh Tokens were also configured. They have a longer lifespan and are used to obtain a new JWT token when the current one expires.

JWTs are compact, URL-safe tokens that encode claims between two parties. A JWT comprises three parts: a header, a payload, and a signature. The header typically specifies the token type (JWT) and the signing algorithm. The payload contains the claims, which are statements about an entity (usually the user) and additional data. The signature, the most crucial part for security, is used to validate the sender's identity and ensure that the message wasn't changed along the way. Moreover, using HTTPS for communications between the client and the server protects the token in transit, preventing man-in-the-middle attacks where a token could be intercepted and reused. Initially, the JWT contained sensitive user information such as the username and email. This was replaced with a random and unique ID generated by Auth0 to prevent the storage of this information on the server.

Backend Implementation

Implementing Auth0 for back-end authentication introduced more complexity. A large overhaul of back-end authentication was necessary, alongside modifications to database structures to adapt to the new approach of not storing sensitive user data in favour of using unique IDs generated by Auth0, ensuring enhanced security and privacy.

The new Authentication Process

The process begins when a user authenticates via Auth0 on the front end, triggering a request from the front end to the back end's /login endpoint. This request includes a JWT granted by Auth0, which gets added to every request to the back-end automatically (for authenticated users). Utilizing the express-oauth2-jwt-bearer package, the back-end verifies this JWT. This verification ensures that the token is valid and the request is authenticated, giving credibility that the JWT contains correct information about the user who sent the request.

Upon successful JWT validation, the back end checks the database for the presence of the associated ID found in the payload. If the ID is found, the system retrieves and sends back relevant user-specific data required after login, such as tutorial files. If the ID does not exist in the database, indicating that a new user has authenticated, the system proceeds to create a new database entry.

For requests to endpoints other than /login, the back end uses the JWT to check if the user is authorized to access the requested endpoint. If the user is not authorized or does not exist, the request is denied.

4.6 Usability Enhancements

4.6.1 Goals

Three goals were identified to improve the usability of the web app. These were identified by analyzing Trylinks 2018's user feedback and some original ideas. Firstly, to improve the landing page a little by centering items, increasing the font, and rewording the description to make explicit what Trylinks aims to do. Secondly, to create a loading screen whilst a user's Links program is compiling, as the compile and deploy pipeline takes a while to load and currently looks like an error page. Thirdly, add functionality for hiding/resizing the code editor window so that users have more space for development.

4.6.2 Justification

While usability is not the main focus of this project, several simple things could be done to bring improvements to Trylinks.

4.6.3 Implementation

Compile and Deploy Pipeline Loading Screen

An issue identified in the previous project was the instability of the compile and deploy pipeline, leading to significant loading times. Initially, users were presented with a fixed-duration prompt during the compilation process. However, this is not a great solution, as it does not always match up with the waiting time. This resulted in users facing prolonged periods of seeing a white screen, and post HTTPS implementation, a "Nginx Not Found" screen, pending the initialization of the Links program on the designated port.

A custom Nginx error screen was implemented on the server to solve this. This was written in plain HTML and CSS and gets displayed when a user enters a subdomain that does not have a Links Process running. The changes can be viewed in the Appendix, Figure B.1.

Resizing of the Tutorial Page Components

The working conditions of Trylinks were quite cramped depending on the display used. Therefore, having taken inspiration from Chapter 2's evaluation, resizeable windows for the tutorial page were implemented. This allows for the resizing or hiding of windows to make more room for the code editor. This includes the tutorial window, code editor, and iFrame window. This was trivially implemented using an angular library.

Slight Improvement of the Landing Page

The landing page was slightly improved to clearly state what Trylinks aims to achieve. This involved coming up with a short catchphrase/description for Links and making the elements more balanced and consistent. The changes can be viewed in the appendix, figure B.2 for those interested.

Chapter 5

Testing & Evaluation

This chapter explores how well the features implemented work in practice and aims to find areas for improvement and future work. The chapter starts by describing the testing and evaluation conducted for each feature implemented in this project in parallel with the Implementation Chapter. Following this, general aspects of the application that were not the main focus of this project are evaluated and tested, such as the performance and cross-browser compatibility.

5.1 Getting Trylinks Running & Updated

The implementation of this feature was a success. Links was upgraded from version 0.72 (Dalry) to 0.97 (Burghmuirhead), granting users access to additional Links features to explore.

Additionally, 'npm audit' was employed to assess the reduction in vulnerabilities following the implementation of this feature. Initially, over 150 vulnerabilities were detected between the client and server, with 20 classified as critical. Currently, no known vulnerabilities remain in the production environment, measured using "npm audit –production", which signifies the successful completion of this feature.

5.2 Redeployment & Improved Development Process

The compile and deploy pipeline was successfully implemented. To evaluate this, a calculation was conducted to determine how much time the pipeline saved for this project.

As expected, manual deployment takes longer. However, considering that this feature took 19.4 hours to implement, it could be argued that instead of saving 7.3 hours, 12.1 hours were lost due to the development time. This was not expected, and whilst this argument holds, it is important to consider other benefits that the CI/CD pipeline provided and the future benefits it will provide:

1. Having this infrastructure set up for future developers who will not be familiar

CI/CD Pipeline Statistics	
Average running time of the pipeline	∼1m52s
Total workflow runs	204
Time spent Deploying (1m52s*204)	∼6.34 hours

Table 5.1: An analysis of the time spent deploying by the CI/CD pipeline.

Manual Deployment Statistics	
Average time to manually deploy	∼4m
Total workflow runs	204
Time spent Deploying (4m*204)	∼13.6 hours

Table 5.2: An analysis of how long manual deployment would have taken.

with deployment is valuable, and time savings will increase over time as more features are implemented.

2. The psychological benefits stemming from the ease of deployment and testing will lead to improved development practises, such as frequent testing and releases.

The git repository was explored to evaluate the effects of code practices. The code practices established at the start of the project successfully helped keep a clean and easy-to-understand git history. Furthermore, git branches have saved development time by allowing us to revert or cancel features easily. However, there were several instances where the practices were not followed. For example, sometimes the commit message format was not followed correctly. At other times, features were incorrectly merged from develop to main (squash instead of rebase and merge), which made some parts of the history confusing. This makes sense, as mistakes happen, and the practices are just guidelines that are not enforced. However, to reduce this in the future, Git Hooks should be implemented to enforce the conventional commit message structure. Additionally, enforced merge strategies were considered, however, GitHub only allows this on the repository level, not the branch level, so this was unsuitable [39].

5.3 Improving the Code Editor

This feature was implemented successfully. The parser used for syntax highlighting supports the essential constructs of Links that beginners are likely to encounter, as identified by one of the Authors. This amounts to 73% (133/181) of the Links constructs. A GitHub issue was created to implement the rest of the constructs[40].

There are now also many additional code editor features enabled by our approach, as highlighted in the Implementation Chapter.

Compared to the OCaml parser, the Lezer grammar approach was found to be significantly slower for larger programs. For an 800-line program, the OCaml implementation parses in 4.1 milliseconds, whereas the Lezer approach parses in 8.1 milliseconds.

However, this is due to the extra work that Lezer puts into setting up the incremental and error-insensitive parsing approach, meaning that in a real use case where parsing and highlighting need to occur on every keystroke, the Lezer Grammar will be faster as only the changed nodes will be prepared, rather than the full 800 lines.

There are also some known downsides to the parsing and highlighting implementation. It tends to be unpredictable sometimes, especially if it encounters a construct that it does not know how to deal with, such as one of the 48/181 that have yet to be implemented. This means that when the parser encounters something like this, it tries to deal with it with the rules that it has, which leads to inaccurate parsing until the parser recovers past the error. This should be fixed with time as the declarative grammar develops and more constructs are supported. Furthermore, it would be beneficial to write more test cases using the Lezer testing framework to ensure that the parser is as close as possible to the OCaml counterpart.

5.4 Improving Security

HTTPS was successfully implemented and now protects user credentials in transit, as requested in the future work section of Trylinks 2018. The implementation of HTTPS was manually tested by ensuring that https://trylinks.net functions correctly and that HTTP traffic gets redirected to HTTPS.

5.5 Improving Authentication

Authentication was successfully improved. Previously, the session length did not persist past an hour, whereas now, it lasts multiple days, and a browser refresh does not invalidate it. Furthermore, the process is faster and more intuitive for users. Instead of requiring 8 mouse clicks to log in a new user, we only need 2-5 depending on whether social login is used.

Trylinks now also complies with many more security and data standards, which can be read about here[37].

However, some downsides were introduced as well. Trylinks now relies on the availability of Auth0, which will prevent users from logging in or authenticating if unavailable, essentially taking all of Trylinks with it. Furthermore, the service will no longer be free past 7500 active users. This was accepted as Trylinks does not have the means to support that many users, as highlighted by the Performance Testing section.

5.6 Usability Enhancements

This set of features was successfully implemented and tested as part of the e2e testing discussed in the Validity section.

5.7 Validity and Cross-browser-compatibility

Testing that Trylinks does what is expected and is compatible with all major browsers was first performed manually using visual inspection. The results table can be viewed in the appendix, Figure 6. After this, the process was automated using end-to-end tests across all major browsers via Cypress [41]. This includes Chrome, Firefox, Edge and Safari (WebKit).

Cypress is an e2e testing framework designed for web applications. It works across multiple browsers and can test anything that runs in a browser. Cypress also allows for visual inspection whilst the tests are running. Additionally, it provides a command-line interface for cross-browser testing, which can be easily implemented in a CI/CD pipeline.

The tests conducted using Cypress were extensive, as we wanted to ensure that all features from Trylinks 2018 have been polished and that the new features implemented are reliable. The dashboard, interactive shell, login page, and tutorial page were tested by writing tests to simulate expected user input into the web app. These tests aimed to confirm if the output produced by the program matched the expected output.

Some things were hard to test exhaustively. For example, the feasibility of testing every combination of inputs a user may enter into the interactive shell was deemed unrealistic. At this point in the project, it was important to balance thorough testing versus the constraints of available resources, so instead, these tests were simplified and focused on more specific requirements. For example, asserting that "The interactive shell is capable of executing, at a minimum, the 12 introductory Links commands and producing accurate outputs". This is done hoping that if these key functionalities operate as expected, it is reasonable to infer that most user interactions will be handled effectively.

End-to-end (e2e) tests have been implemented for all six Links Web Programming tutorials. The testing process usually flows as follows: Initially, the tutorial's path is visited. Then, the existing content within the code editor is replaced with the solution for the tutorial. After this, the compile button is activated, and the iFrame content is tested to ensure successful compilation. Verification methods vary, ranging from straightforward, such as identifying a "hello world" paragraph in the iFrame's body, to more complex, which involves interacting with the iFrame to add an item to the database. Similar tests were written for the interactive shell, login, and dashboard page.

An example of a test written for Trylinks using the Cypress testing framework and a look at the visual workflow possible is available in the Appendix, figures A.1 and A.2 for those interested. Elements that could not be automated due to time and skill restraints were verified using visual inspection (e.g. resizeable code editor). A detailed report can be viewed in the Appendix (Table A.2), which lists each test performed.

A significant bug was found after conducting this testing, and two observations were discovered:

1. If the same tutorial is recompiled and redeployed many times, the running times will be good and stable. They range between 3s-8s, depending on the tutorial.

2. When a user switches from one tutorial to another, the running time increases exponentially until it is no longer usable or the website crashes.

It is hard to be sure whether this is what the Author of Trylinks 2018 meant by the unstable pipeline they mentioned, or if this is caused by updating the dependencies and not properly changing the code to match the new behaviours. However, after some debugging, the management of WebSocket connections on the server was identified as the problem. It turns out that listeners were accumulating on the backend of Trylinks, as the code did not remove listeners when they were no longer needed. This means that with each compile attempt, an additional listener was established on the server, leading to the user's program being compiled repeatedly, matching the total number of compilations initiated by the user. By the fifth compilation, the user's program would enter a loop, being compiled 5 times over, significantly impacting the system's efficiency.

After fixing this bug, subsequent tests verified that the system works smoothly across all modern browsers with great performance. User programs now compile in less than 5 seconds, interactive shells start-up in no more than 2 seconds, and commands are returned to the user in less than 0.25 seconds. Additionally, logging in takes at most 1 second from submitting details.

Despite these successes, it's important to acknowledge the limitations of our testing approach. Firstly, the testing strategy was limited. It primarily focuses on verifying that the application behaves as expected with typical user inputs. A broad spectrum of testing strategies, such as boundary testing, unit/integration testing and fuzz testing, remain untested. Furthermore, while we have verified the project's functionality under no load, we cannot be sure it remains functional under varied conditions. This concern is partially addressed in the performance testing section.

5.8 Performance

Initially, JMETER tests were planned to assess the app's performance under heavy user load. However, due to the complexity of setting up such tests, especially with mocking Auth0 authentication and the difficulty in testing non-string outputs like compiled Links applications, this approach was deemed infeasible within our project's timeframe. Additionally, since JMETER focuses on server load, alternative methods for client-side testing would still need to be investigated.

Instead, the decision was made to utilize the existing Cypress tests described in the cross-browser-testing section, which can test both the client and server. Although Cypress is not intended for load testing, running multiple Cypress instances in parallel could accurately simulate concurrent user interactions, assessing the app's ability to handle multiple users. The intention was to scale these tests across multiple machines and run them in parallel using an online tool. However, a custom solution had to be built as the online load testing tools researched usually required working with endpoints or a bespoke testing language. The one tool that did support Cypress tests, Step [42], unfortunately, worked only for an older Cypress version and was not free past a trial.

Instead, 50 Auth0 test accounts were created on the dev domain, and a script was written that will run up to 10 instances of the Cypress tests in parallel. Finally, these scripts can be executed on various machines simultaneously, and the load can be varied to see how the running time or pass/fail statistics vary under load. The testing ranged from 1 user across 1 computer all the way to partial failure at 20 users across 4 computers.

The total time to complete the Cypress tests was measured against the number of simultaneous users. To complete the 12 tutorial snippets in the interactive shell, the times ranged from 17 seconds (1 user) to partial failure (20 users). Similarly, completing all 5 tutorials ranged from 58 seconds (1 user) to partial failure (20 users). By partial failure, we mean that Cypress tests cannot pass due to taking over 2 seconds to initiate an interactive shell and over 8 seconds to compile a Links web app.

These tests revealed that the bottlenecks are the spawning of interactive shells and the compilation of Links processes. Therefore, to get more useful results, we varied the number of simulated users and then tested how long the compilation and spawning of a shell takes until it is no longer reasonable, or the website crashes. The results can be viewed below.

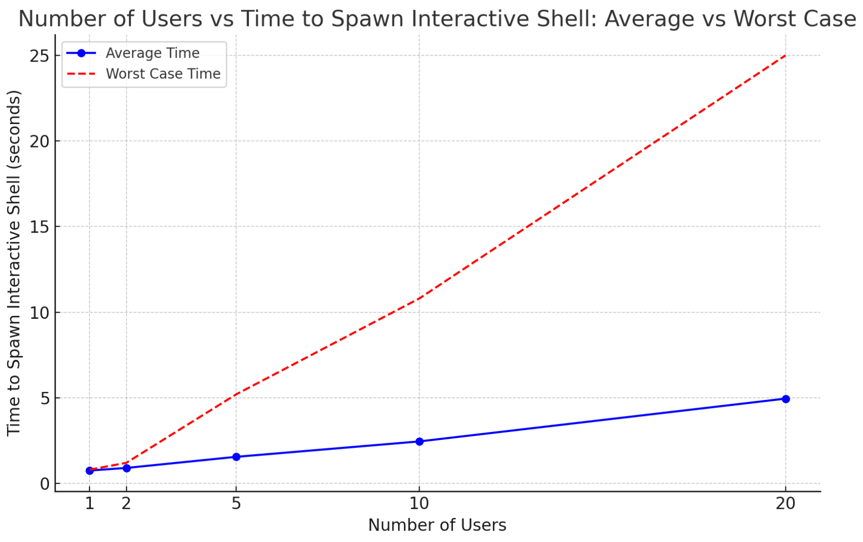


Figure 5.1: A graph of the number of users spawning shells against shell spawn time

As highlighted in Figure 5.1, the average time for the interactive shell to spawn is acceptable until 20 users, after which it starts to feel slow. The deterioration in the worst-case response times accelerates at a worse pace. It is most likely linked to how many interactive shells are being initiated at the same time, as once initiated, they tend to have a lower effect on the performance, even with many users.

The tutorial page seems to reach the failure point more quickly. At 20 concurrent users, the compilation time is still reasonable; however, the whole web app slows significantly. Even simple tasks, such as loading the tutorial content from the database, start slowing down, and instead of instantly fetching the data, users are left with an empty screen for up to 10 seconds. Upon further investigation, it was discovered that at this point, the

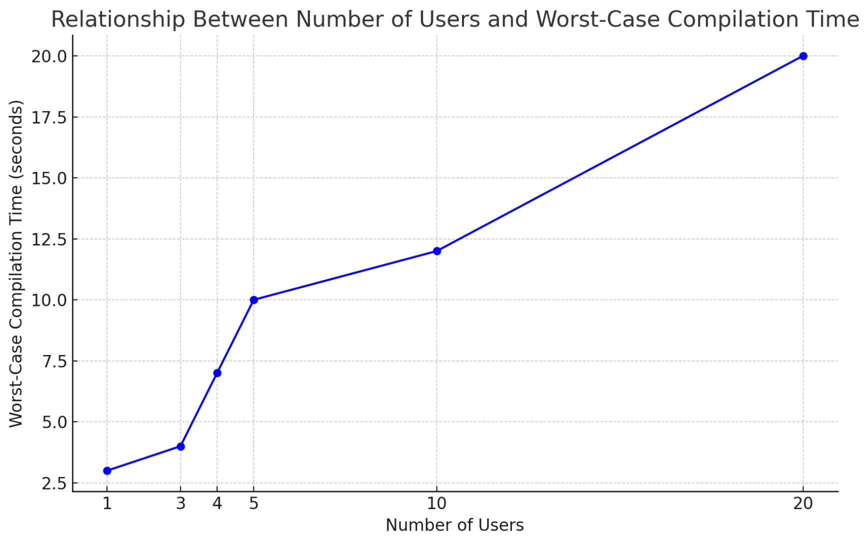


Figure 5.2: A graph of the number of users compiling tutorials against compilation time.

CPU and Memory Usage of the Digital Ocean server reached 100%, signifying that the load generated by 20 users, each attempting to compile 5 Links programs, surpasses the processing capabilities of the allocated server resources.

This shows that Trylinks has come a long way. Initially, Trylinks struggled to compile 5 Links web programs for one account, but it can now do it for around 20. However, clearly, this can still be improved upon.

The machine used for deployment is quite weak, only containing 2GB of RAM and 1 vCPU, so upgrading this when needed is one method to support more users. However, even with the current machine, better results could likely be achieved using the containerization system discussed in Chapter 2. This would allow us to limit the resources consumed by each container, which may slow the overall compilation process but prevent the compilation from taking up too many resources and slowing other parts of the web app. Another option may be to limit the number of programs that are compiled at once and put users in a queue. These improvements will be left as future work.

5.9 Known Issues & Bugs

Throughout the Testing and Evaluation chapter, several bugs have been identified.

Firstly, the already mentioned parsing and syntax highlighting unreliability should improve with more time and some extra development.

Secondly, the performance of the app under load, which starts to struggle at 20 concurrent users. Ideas for improvement have been touched upon in Chapter 2 and are given as future work in the Conclusion Chapter.

Chapter 6

Conclusion

6.1 Retrospective

6.1.1 Revisiting the Goals

At the beginning of the project, we started with an outdated and non-functional implementation of Trylinks. A thorough evaluation was performed, and the following goals were identified:

1. To upgrade Trylinks to more recent technological standards, remove vulnerabilities, and redeploy Trylinks.
2. To streamline the development process, with a focus on minimizing the risk of the project becoming outdated again.
3. To implement as many incremental improvements/features as time allows.

These goals were successfully achieved within the time frame for this project. They were tested, verified, and evaluated using tools and techniques like e2e testing and visual inspection. This signifies the successful completion of this project.

6.1.2 Obstacles

Some obstacles were encountered during the development of this project.

The first thing that stands out is that the parser took around 35 hours to implement, excluding research and project meetings. The parser is an important part of Trylinks, as it enables syntax highlighting and allows for a much simpler implementation of additional features. However, it would have been a good idea to grasp how long the parser would take before implementation and consider if those hours could be spent on more than one feature that could have a greater impact on the users.

The second challenge was a recurring one throughout this project. This is the unexpected difficulty of building upon another developer's work. Often, straightforward tasks, such as implementing HTTPS or a new authentication system, took much longer to implement due to the need to integrate with existing code which frequently led to unforeseen complications. In contrast, creating a website from scratch allows thorough planning to ensure compatibility, thus smoothing the implementation process.

Finally, a very large amount of time was spent completing tasks on the server side. Examples include deployment, HTTPS setup, CI/CD pipeline implementation, and specialised Nginx configuration. This was compounded by the fact that it was done on the dev domain first, and was repeated for the prod domain close to the deadline for this project. This brought about unnecessary stress, as it reduced confidence in the functionality of the main domain due to reduced testing time compared to dev. It would have been good to consider that even with a CI/CD pipeline and identical code on the dev and prod domains, it is important to test the prod domain just as carefully due to server differences.

6.1.3 Limitations

User feedback was not utilized in this project. This is partly due to the availability of previous user feedback conducted for Trylinks 2018, which gave ideas for missing project features and verified user satisfaction with the base system. Therefore, it was assumed that since the project aims to revitalize Trylinks and not focus much on usability, user feedback will not be necessary as there will not be much to ask users. Enhancements such as the CI/CD Pipeline, improved security measures, a new authentication process, and GitHub practices were either unrelated to the end-users, like the CI/CD Pipeline, or were more suitably quantified. For instance, asking users to compare session lengths, ease of login, and security benefits was deemed unrealistic, and it would be too difficult to gain enough user feedback to perform meaningful A/B tests.

Despite this, upon reflection, it would have been satisfying to obtain metrics on user satisfaction for Trylinks 2023 in addition to those from the 2018 project. There is also a potential that issues were introduced that were previously not there. The chances of this was increased at the end of the project, as extra time was available and a few usability improvements were implemented. However, it was too late to collect user feedback at this point.

Secondly, this project did a lot to upgrade Trylinks. For example, we now have a more advanced code editor and the latest Links version. However, the absence of new tutorials that use the capabilities of the updated Links version means we are not fully capitalizing on this improvement. Currently, users must be aware of the new features to utilize, limiting the potential benefits these updates could offer. This is discussed further in the future work section.

6.1.4 Final Summary

A final summary of the work conducted for this project is below:

Revitalize Trylinks:

- The vulnerabilities in Trylinks were addressed by upgrading Angular by many major versions and updating or replacing out-of-date packages. Additionally, Links was updated to the latest version. Finally, a Digital Ocean server was secured, and Trylinks was redeployed on trylinks.net and dev.trylinks.net.

Facilitate Future Development:

- A conventional commit and branching strategy was adopted. To further accommodate this, the main and develop branches were locked to direct commits.
- A CI/CD workflow was established for Trylinks. This enables automatic deployment when developers merge their code with the production or development branches on GitHub. The develop branch is served at dev.trylinks.net, while the production branch is accessible at trylinks.net.
- Automated cross-browser tests were created.

Add Incremental Improvements:

- Many enhancements were made to the code editor, notably the addition of syntax highlighting, achieved through implementing a parser.
- Security enhancements were achieved through the implementation of HTTPS.
- A new authentication system was introduced, utilizing Auth0 to delegate user management and data storage to an identity provider (idP).
- In response to user feedback, the code editor was made resizable.
- A clearer loading screen was implemented for displaying the status when users' Links programs are being compiled.
- The home page underwent a slight restyling to enhance user understanding of what Trylinks offers.
- Some of the instabilities of the Compile and Deploy pipeline were fixed mentioned the Testing & Evaluation Chapter.

6.2 Future Work

6.2.1 Additional Tutorials & Evaluation System Integration

The previous Trylinks project mentioned that, "One of the greatest sources of confusion reported in using TryLinks is that users do not always know the end goals due to the lack of an evaluation system."[2]. Future work could focus on integrating this feature into the main codebase and creating tutorials for the new Links features since 2018.

6.2.2 Further code-editor improvements

The Code Mirror 6 editor, implemented along with the Lezer Grammar, can facilitate many additional code editor features. Future work could focus on finishing the parser to support the rest of the Links constructs and then use it to implement additional code-editor features, such as auto-completion, auto-indentation, and code snippets.

Additionally, automated testing of the Lezer Grammar could be implemented. This would involve creating Links test cases within the official Lezer Grammar testing framework[43]. This could then be put through the OCaml parser, and the output can be processed into Lezer Parser format. If this is used as the expected output to the test cases, it will enable consistent and systematic verification of the Grammar with each update to Links on the server.

6.2.3 A better compile and deploy approach

As discussed in the Evaluation of Previous Work chapter, the state-of-the-art method for arbitrary code deployment seems to be containerization. Transitioning to this method from directly compiling Links programs on the server could greatly improve the stability of the Compile and Deploy pipeline.

6.2.4 Improvement of UI and Beta Testing

Both Trylinks 2018 and Trylinks 2023 did not focus much on usability or UI design. A project on creating a nice UI for Trylinks using a tool like Figma, along with comprehensive user testing for said design, would be satisfying and could give the website a more professional feel.

6.2.5 Admin System

As stated in Trylinks 2018[2], an admin system could benefit the Links development team, as they could add new tutorials when Links upgrades with new features without having to interact with the code-base.

Bibliography

[1] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web Programming Without Tiers. `https://homepages.inf.ed.ac.uk/slindley/papers/links-fmco06.pdf`, 2006.

[2] Nick Wu. An interactive online platform to learn the links programming language. `https://links-lang.org/papers/undergrads/ug4_20181051.pdf`, 2018.

[3] Nick Wu. Trylinks-2018: Future work. `https://links-lang.org/papers/undergrads/ug4_20181051.pdf`, 2018. Page 74.

[4] James Cheney. Trylinks 2023 project proposal. `https://dpmt.inf.ed.ac.uk/ug4/project/7215`, 2023.

[5] Design Dev. The evolution of web development, 2023. Available from LinkedIn: `https://www.linkedin.com/pulse/evolution-web-development-designdevbiz/`.

[6] Chris Don. Try haskell! an interactive haskell tutorial in your browser. `https://tryhaskell.org/`.

[7] Codecademy: Interactive coding tutorials. `https://www.codecademy.com/`.

[8] The official links installation documentation. `https://github.com/links-lang/links/blob/master/INSTALL.md`.

[9] Nick Wu. Trylinks2018: Complicated links setup. `https://links-lang.org/papers/undergrads/ug4_20181051.pdf`, 2018. Page 18.

[10] Links-Lang Project Contributors. Database setup documentation for links. `https://github.com/links-lang/links/wiki/Database-setup`.

[11] Joe Dibley. Nist password guidelines. Netwrix Blog, Nov 2022.

[12] Authentication cheat sheet: Login throttling. OWASP Cheat Sheet Series.

[13] Google / Harris Poll. Google security infographic: Online security survey. `https://services.google.com/fh/files/blogs/google_security_infographic.pdf`, February 2019.

[14] Lani Leuthvilay, HYPR. Hypr's 2-year password usage study examines human behavior. `https://blog.hypr.com/hypr-password-study-findings`, December 2023.

[15] Sunwall, Evan and Neusesser, Tim. Error-message guidelines. `https://www.nngroup.com/articles/error-message-guidelines/`, February 2024.

[16] Russell, Joseph. App login design: Choosing the right user login option for your app. `https://uxmag.com/articles/app-login-design-choosing-the-right-user-login-option-for-your-app`, September 2022.

[17] Authentication cheat sheet: Session management. OWASP Cheat Sheet Series.

[18] Nasson, Randy. Balance user experience and security to retain customers. `https://auth0.com/blog/balance-user-experience-and-security-to-retain-customers/`, August 2020.

[19] Official docker documentation: Getting started. `https://docs.docker.com/get-started/overview/`, November 2023.

[20] Atlassian. Deployment automation: What is it. `https://www.atlassian.com/devops/frameworks/deployment-automation`.

[21] Nick Wu. Trylinks client repository. `https://github.com/NickWu007/TryLinks-Client-v2`, 2019.

[22] Nick Wu. Trylinks server repository. `https://github.com/NickWu007/TryLinks-Server`, 2018.

[23] What are conventional commits: A quick summary. `https://www.conventionalcommits.org/en/v1.0.0/#summary`.

[24] Angular update guide. `https://update.angular.io/`.

[25] Gharaati, Amirhosein. Mastering git: The power of conventional commit messages. `https://blog.stackademic.com/mastering-git-the-power-of-conventional-commit-messages-1bfbd1cae2c2`, September 2023.

[26] An example of a conventional commit message. `https://www.conventionalcommits.org/en/v1.0.0/`.

[27] Digitalocean. `https://www.digitalocean.com/`.

[28] Nginx. `https://www.nginx.com/resources/glossary/nginx/`.

[29] Slava Vaniukov. Code editor use statistics. `https://www.softermii.com/blog/top-ides-for-software-development`, 2022.

[30] Sarkar, Advait. The impact of syntax colouring on program comprehension. `https://ppig.org/files/2015-PPIG-26th-Sarkar1.pdf`, 2015. Page 1, Line 11.

[31] Marijn Haverbeke. Challenges of parsing inside the context of code editor. `https://marijnhaverbeke.nl/blog/lezer.html#:~:text=It%20also%20generates%20very%20hefty,heavily%20inspired%20by%20tree%2Dsitter.`, 2019. Lines 17-27.

[32] The tree-sitter algorithm. `https://tree-sitter.github.io/tree-sitter/`.

[33] Stefan Münnich. Codemirror component (adapted). `https://github.com/webern-unibas-ch/awg-app/blob/a142590a796a0cef8453db5b91f300a5d491687b/src/app/shared/codemirror/codemirror.component.ts#L8`, 2022.

[34] Samuel Höra. Codemirror component (original). `https://github.com/robotcoral/ngx-codemirror6/blob/main/src/codemirror.component.ts`, 2021.

[35] Catherine Chipeta. What is https. `https://www.upguard.com/blog/what-is-https`, 2023.

[36] Auth0 pricing. `https://auth0.com/pricing`.

[37] Auth0. Auth0 compliance & certifications. `https://auth0.com/docs/secure/data-privacy-and-compliance`.

[38] Dan Arias. The complete guide to angular user authentication with auth0. `https://auth0.com/blog/complete-guide-to-angular-user-authentication/`, November 2022.

[39] bk2204. Merging branches on github via pull requests with enforced merge strategies? `https://stackoverflow.com/questions/61897501/is-there-a-way-to-merge-branches-on-github-via-pull-requests-with-different-and#:~:text=GitHub%20lets%20you%20adjust%20merge,option%20is%20to%20squash%20merge`, April 2022.

[40] Mantas Maciulis. Github issue to implement the remaining links constructs. `https://github.com/mantasmaciulis/TryLinks-2023/issues/39`.

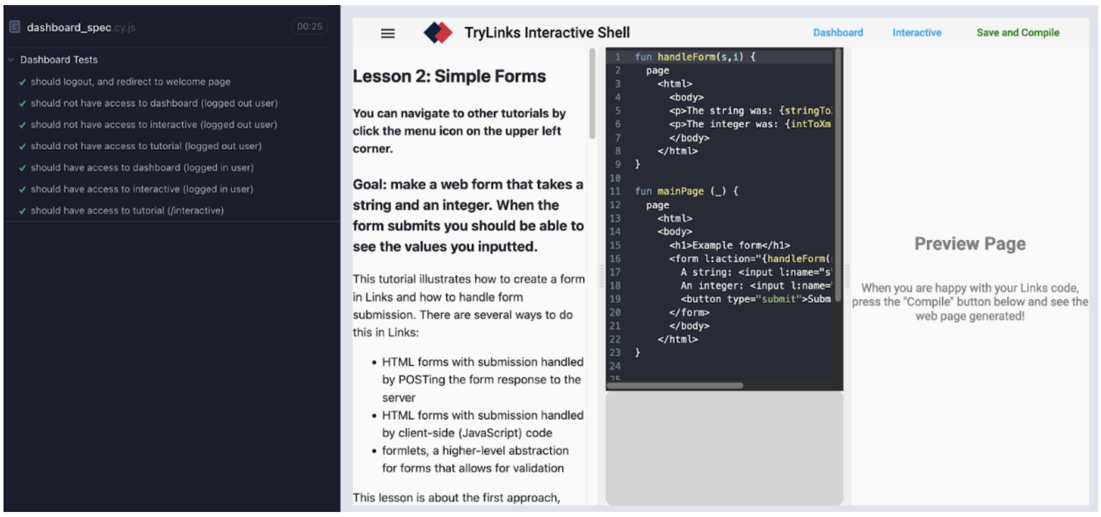
[41] Eran Kinsbruner. What is cypress testing. `https://www.perfecto.io/blog/cypress-testing`, 2021.

[42] Cypress load testing with step. `https://step.exense.ch/`.

[43] Lezer test library example. `https://lezer.codemirror.net/examples/test/`.

Appendix A

Testing & Evaluation Documents


dashboard_spec.cy.js 00:25

Dashboard Tests

- ✓ should logout, and redirect to welcome page
- ✓ should not have access to dashboard (logged out user)
- ✓ should not have access to interactive (logged out user)
- ✓ should not have access to tutorial (logged out user)
- ✓ should have access to dashboard (logged in user)
- ✓ should have access to interactive (logged in user)
- ✓ should have access to tutorial (/interactive)

TryLinks Interactive Shell

Lesson 2: Simple Forms

You can navigate to other tutorials by click the menu icon on the upper left corner.

Goal: make a web form that takes a string and an integer. When the form submits you should be able to see the values you inputted.

This tutorial illustrates how to create a form in Links and how to handle form submission. There are several ways to do this in Links:

- HTML forms with submission handled by POSTing the form response to the server
- HTML forms with submission handled by client-side (JavaScript) code
- formlets, a higher-level abstraction for forms that allows for validation

This lesson is about the first approach,

```
fun handleForm(s,i) {
  page
    <html>
      <body>
      <p>The string was: {stringTo
      <p>The integer was: {intToXm
      </body>
    </html>
}

fun mainPage (_) {
  page
    <html>
    <body>
      <h1>Example form</h1>
      <form l:action="{handleForm(
        A string: <input l:name="s
        An integer: <input l:name=
        <button type="submit">Subm
      </form>
      </body>
    </html>
}
```

Preview Page

When you are happy with your Links code, press the "Compile" button below and see the web page generated!

Figure A.1: A screenshot showing the cypress workflow with visual inspection and automated testing.

```
beforeEach(function() {
    // Log in before each test in this suite
    cy.loginTestUser()
  });

it('should redirect user back to the dashboard', function() {
  cy.get('button:contains("Launch Links Interactive Mode")').click();
  cy.get('.tl-dashboard-button').click();
  cy.url().should('eq', 'https://dev.trylinks.net/dashboard');
});
```

Figure A.2: An example of a simple cypress test.

Browser	**Spec**	**Status**
Chrome	Dashboard Test	All Pass
Chrome	Interactive Tests	All Pass
Chrome	Authentication Tests	All Pass
Chrome	Tutorial Tests	Initial Fail, Now Pass
Firefox	Dashboard Test	All Pass
Firefox	Interactive Tests	All Pass
Firefox	Authentication Tests	All Pass
Firefox	Tutorial Tests	Initial Fail, Now Pass
Safari	Dashboard Test	All Pass
Safari	Interactive Tests	All Pass
Safari	Authentication Tests	All Pass
Safari	Tutorial Tests	Initial Fail, Now Pass
Microsoft Edge	Dashboard Test	All Pass
Microsoft Edge	Interactive Tests	All Pass
Microsoft Edge	Authentication Tests	All Pass
Microsoft Edge	Tutorial Tests	Initial Fail, Now Pass

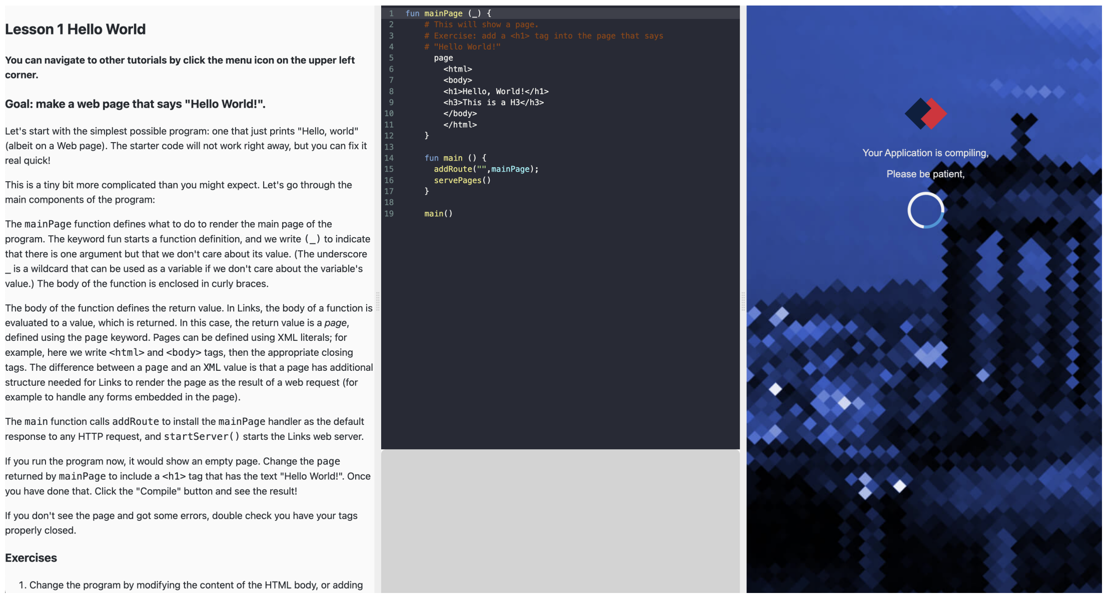
Table A.1: Test Status by Browser for Automatic Cypress Tests

Visual Inspection Test	**Status**
Session Persists for 3 Days	Pass for all browsers
Tutorial Page Windows are Resizeable	Pass for all browsers
Interactive Mode Visually Resembles a Shell	Pass for all browsers
Tutorials Correctly Link GitHub Solutions	Pass for all browsers
Tutorials Instructions Correctly Styled	Pass for all browsers

Table A.2: Test Status for Visual Inspection Across All Browsers

Appendix B

Trylinks 2023 Screenshots



Lesson 1 Hello World

You can navigate to other tutorials by click the menu icon on the upper left corner.

Goal: make a web page that says "Hello World!".

Let's start with the simplest possible program: one that just prints "Hello, world" (albeit on a Web page). The starter code will not work right away, but you can fix it real quick!

This is a tiny bit more complicated than you might expect. Let's go through the main components of the program:

The `mainPage` function defines what to do to render the main page of the program. The keyword fun starts a function definition, and we write `(_)` to indicate that there is one argument but that we don't care about its value. (The underscore _ is a wildcard that can be used as a variable if we don't care about the variable's value.) The body of the function is enclosed in curly braces.

The body of the function defines the return value. In Links, the body of a function is evaluated to a value, which is returned. In this case, the return value is a *page*, defined using the `page` keyword. Pages can be defined using XML literals; for example, here we write `<html>` and `<body>` tags, then the appropriate closing tags. The difference between a `page` and an `XML` value is that a page has additional structure needed for Links to render the page as the result of a web request (for example to handle any forms embedded in the page).

The `main` function calls `addRoute` to install the `mainPage` handler as the default response to any HTTP request, and `startServer()` starts the Links web server.

If you run the program now, it would show an empty page. Change the `page` returned by `mainPage` to include a `<h1>` tag that has the text "Hello World!". Once you have done that. Click the "Compile" button and see the result!

If you don't see the page and got some errors, double check you have your tags properly closed.

Exercises

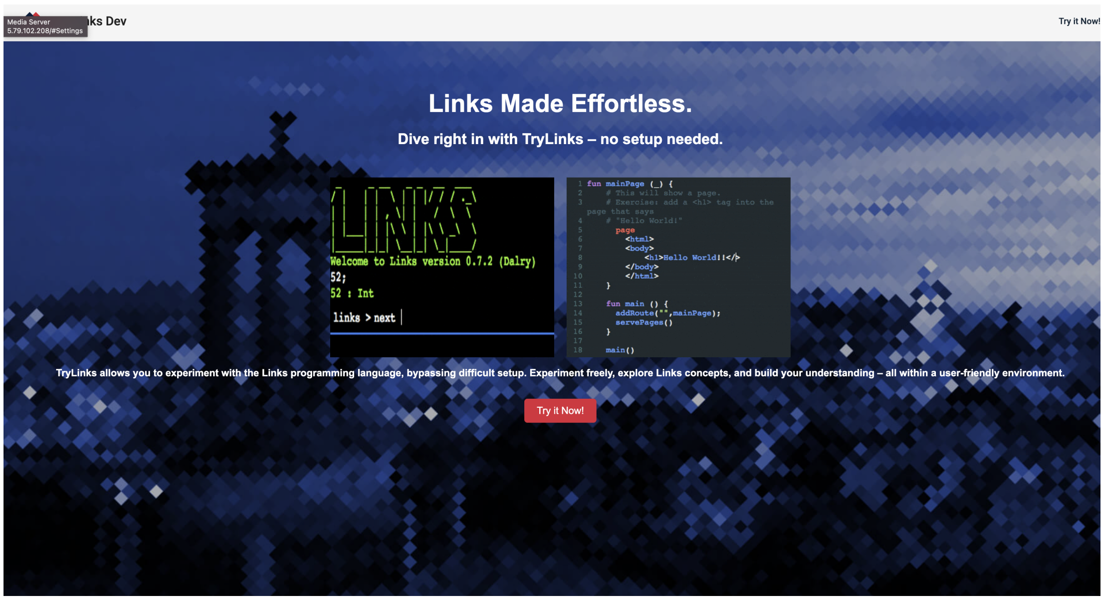
1. Change the program by modifying the content of the HTML body, or adding

```
fun mainPage (_) {
    # This will show a page.
    # Exercise: add a <h1> tag into the page that says
    # "Hello World!"
      page
        <html>
        <body>
        <h1>Hello, World!</h1>
        <h3>This is a H3</h3>
        </body>
        </html>
    }

    fun main () {
      addRoute("",mainPage);
      servePages()
    }

    main()
```

Your Application is compiling,

Please be patient,

Figure B.1: A screenshot highlighting the new loading page.



Links Made Effortless.

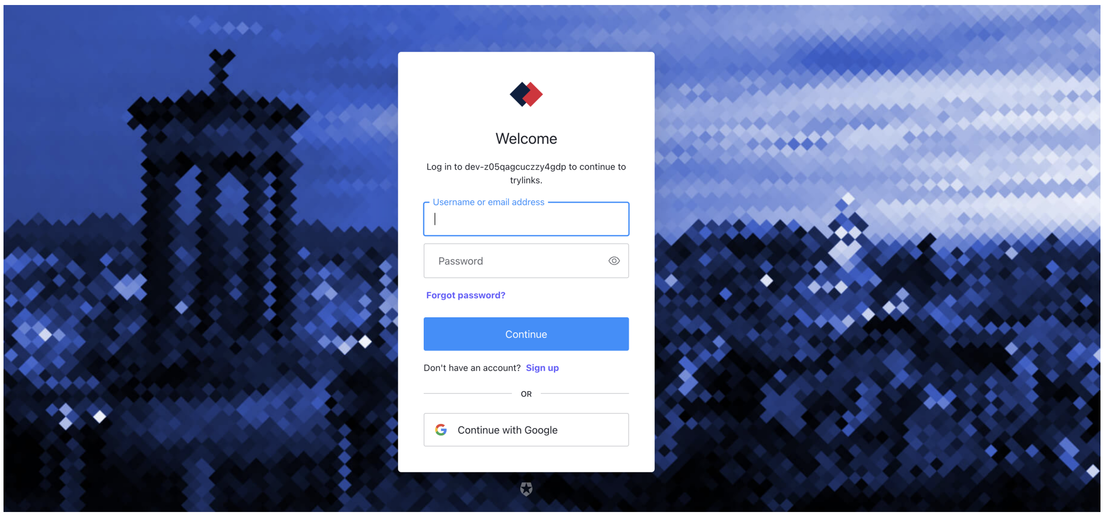
Dive right in with TryLinks – no setup needed.

```
Welcome to Links version 0.7.2 (Dalry)
52;
52 : Int
links > next
```

```
fun mainPage (_) {
    # This will show a page.
    # Exercise: add a <h1> tag into the
page that says
    # "Hello World!"
      page
        <html>
        <body>
            <h1>Hello World!!</>
        </body>
        </html>
    }

    fun main () {
      addRoute("",mainPage);
      servePages()
    }

    main()
```

TryLinks allows you to experiment with the Links programming language, bypassing difficult setup. Experiment freely, explore Links concepts, and build your understanding – all within a user-friendly environment.

Figure B.2: A screenshot of the new homepage.

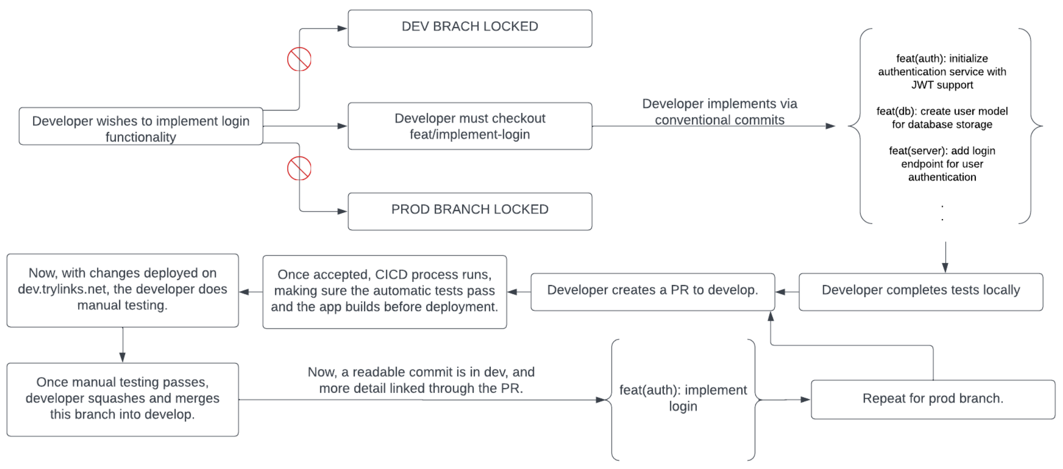

Welcome

Log in to dev-z05qagcuczzy4gdp to continue to trylinks.

Figure B.3: A screenshot of the new login page.

Appendix C

Miscellaneous



DEV BRACH LOCKED
Developer wishes to implement login functionality
Developer must checkout feat/implement-login
PROD BRANCH LOCKED
Developer implements via conventional commits
feat(auth): initialize authentication service with JWT support
feat(db): create user model for database storage
feat(server): add login endpoint for user authentication
Developer completes tests locally
Developer creates a PR to develop.
Once accepted, CICD process runs, making sure the automatic tests pass and the app builds before deployment.
Now, with changes deployed on dev.trylinks.net, the developer does manual testing.
Once manual testing passes, developer squashes and merges this branch into develop.
Now, a readable commit is in dev, and more detail linked through the PR.
feat(auth): implement login
Repeat for prod branch.

Figure C.1: A visual diagram illustrating the planned development process.

```
var db = database "todo";
var items = table "items" with (name : String) from db;

mutual {
  fun showList() server {
   page
    <html>
     <body>
      <form l:action="{add(item)}" method="POST">
        <input l:name="item"/>
        <button type="submit">Add item</button>
      </form>
      <table>
       {for (item <- query {for (item <-- items) [item]})
          <tr><td>{stringToXml(item.name)}</td>
              <td><form l:action="{remove(item.name)}" method=
   "POST">
                   <button type="submit">Done</button>
```

```
                  </form>
              </td>
          </tr>}
       </table>
      </body>
    </html>
  }

  fun add(name) server {
   insert items values [(name=name)];
   showList()
  }

  fun remove(name) server {
   delete (r <-- items) where (r.name == name);
   showList()
  }
}

showList()
```

Listing C.1: An Example of a Links Program