My Learning Journey

Salman Almuzaini[1]

Business or Academic Affiliation's Full Name 1, City, State, Zip Code, Country

This document is a detailed record of my foray into data science with Rust, capturing the essence of the journey from initial setup using Cargo to tackling various challenges. It narrates the progression through stages that involved integration with APIs, meticulous error management, and navigating Docker for environmental stability, emphasizing the intricacies of cross-compiling and static linking.

Throughout this exploration, data played a central role, with CSV files such as "Table 2.4.4U. Price Indexes for Personal Consumption Expenditures by Type of Product" and "Table 2.6. Personal Income and Its Disposition, Monthly" serving as fundamental resources. These files, derived from the Bureau of Economic Analysis (BEA), were crucial in analyzing economic indicators within the Rust environment.

A notable point in the journey was the division of the "Personal Consumption Expenditures Price Index.xlsx" file into individual CSV sheets such as "Contents.csv", "Table 1.csv", through "Table 7.csv", a process facilitated by a Python script sourced from GitHub. This script, `getsheets.py`, was instrumental in transforming the multifaceted Excel document into a structured array of CSV files, thereby simplifying the dataset for subsequent analysis.

The narrative also highlights the strategic shift to a more manageable dataset, 'organization-100.csv', for effective analysis, and the utilization of 'iris.csv' for exploring data processing capabilities within Rust. Additionally, the Rust Jupyter notebook, sourced from the Rust user's forum and adapted from online tutorials, provided an interactive platform for executing and testing code snippets, further enriching the learning experience.

The document reflects on the importance of adaptability in learning and the precision required for robust error handling in Rust. It is a testament to the technical milestones achieved and an introspection of the strategic decision-making and problem-solving approaches vital for data science within the Rust ecosystem.

References

[1] Bureau of Economic Analysis, "Table 2.4.4u. Price Indexes for Personal Consumption Expenditures by Type of Product", 2020.. Retrieved 7 November 2023. https://apps.bea.gov/iTable/?reqid=19&step=3&isuri=1&select_all_years=0&nipa_table_list=2013&series=m&first_year=2020&last_year=2022&scale=-99&categories=underlying&thetable=

[2] Bureau of Economic Analysis, "Table 2.4.5u. Personal Consumption Expenditures by Type of Product", 2020.. Retrieved 7 November 2023. https://apps.bea.gov/iTable/?reqid=19&step=3&isuri=1&select_all_years=0&nipa_table_list=2014&series=m&first_year=2020&last_year=2022&scale=-99&categories=underlying&thetable=

[3] Bureau of Economic Analysis, "Table 2.4.6u. Real Personal Consumption Expenditures by Type of Product, Chained Dollars", 2023.. Retrieved 7 November 2023. https://apps.bea.gov/iTable/?reqid=19&step=3&isuri=1&select_all_years=0&nipa_table_list=2014&series=m&first_year=2020&last_year=2022&scale=-99&categories=underlying&thetable=

[4] Bureau of Economic Analysis, "Table 2.6. Personal Income and Its Disposition, Monthly", 2023.. Retrieved 7 November 2023. https://apps.bea.gov/iTable/?reqid=19&step=3&isuri=1&1921=survey&1903=76

[5] Bureau of Economic Analysis, "Table 2.8.7. Percent Change from Preceding Period in Prices for Personal Consumption Expenditures by Major Type of Product, Monthly", 2023.. Retrieved 7 November 2023. https://apps.bea.gov/iTable/?reqid=19&step=3&isuri=1&1921=survey&1903=84

[1] Insert Job Title, Department Name, and AIAA Member Grade (if any) for first author.

[6] Bureau of Economic Analysis, “Personal Consumption Expenditures Price Index”, 2023.. https://www.bea.gov/data/income-saving/personal-income

[7] Ming, S., “Getsheets.py”, 2021.. Retrieved 7 November 2023. https://gist.github.com/scottming/99c09685360376d4cac2de7c891e8050/9670acd5044d11871188e04b8123263ad132f512

[8] Unknown, “Iris.csv”, 2023.. Retrieved 7 November 2023. https://gist.github.com/netj/8836201

[9] Unknown, “Organization-100.csv”, 2023.. Retrieved 7 November 2023. https://www.datablist.com/learn/csv/download-sample-csv-files

[10] Unknown, “Rust Jupyter Notebook”, 2023.. Retrieved 7 November 2023. https://datacrayon.com/data-analysis-with-rust-notebooks/setup-anaconda-jupyter-and-rust/

[11] Studies, K. A. P., and Center, R., “KAPSARC's API Portal”, 2023.. https://datasource.kapsarc.org/api/explore/v2.1/console

A. Day 1: Rust, APIs, and Containerization Challenges

Embarking on a journey to master Rust for data science, I set my sights on leveraging APIs for dynamic data retrieval. The KAPSARC (King Abdullah Petroleum Studies and Research Center) API promised a wealth of energy data, but tapping into this resource proved to be a complex task.

1. *Setting Up with Cargo.toml*

The first order of business was configuring the Cargo.toml to declare the dependencies necessary for making HTTP requests (reqwest) and handling JSON data (serde). After overcoming an initial hiccup where reqwest wasn’t recognized, a corrected Cargo.toml looked like this:

```toml
[dependencies]
reqwest = "0.11"
serde = "1.0"
serde_json = "1.0"
```

This setup was my gateway to the world of Rust dependencies.

2. *The Build: Rust’s Compilation Dance*

Executing cargo build set off a cascade of downloads and compilations, a testament to the rich ecosystem Rust offers. Watching libraries like cfg-if and percent-encoding get ready for action was my first taste of victory.

3. *The Runtime: A Tryst with Errors*

However, my celebration was short-lived as cargo run threw a stark runtime error — a missing file. It underscored the importance of crafting precise error handling in Rust. Adjusting the error handling logic, I braced myself for the next attempt.

But the hurdles continued. The API’s JSON response was missing the total_count field, expected by my parsing logic. Rust’s Option type came to the rescue, allowing me to handle the unpredictability of real-world data:

```rust
#[derive(Deserialize)]
struct ApiResponse {
    total_count: Option<i32>, // Now optional, no missing field error
    // Other fields as required
}
```

This adjustment was a key learning moment in data flexibility and error resilience.

4. *Docker: The Container Conundrum*

As the project progressed, I decided to encapsulate the development environment using Docker, with Alpine Linux as the base image for its minimalistic footprint. This decision led to a confrontation with a new beast: building Rust applications for musl libc instead of the default glibc. This choice, aimed at creating a more secure and portable executable, introduced me to the challenges of cross-compiling Rust applications.

The openssl-sys crate, necessary for HTTPS requests, became the center of my struggle as it clashed with the musl-based environment. The errors were obscure, leading to a deep dive into Alpine's package manager and Rust's compilation flags. The learning curve was steep, as I navigated through the intricacies of static linking and system dependencies.

5. Project Scope and Reflections

Amidst the technical battles, the scope of my project began to shift. I had to reassess my goals and adapt my approach to align with my growing understanding of Rust's capabilities and limitations. The initial intention to use Jupyter notebooks for an interactive Rust experience was shelved in favor of focusing on the API integration aspect.

Day 1 was not just about code; it was about adaptability, problem-solving, and continuous learning. It reinforced the iterative nature of programming, where each error leads to a deeper insight, and every decision shapes the trajectory of the project.

In the rearview mirror, the day was a confluence of technical challenges, strategic pivots, and the inevitable realization that learning Rust was as much about understanding its ecosystem as it was about the language constructs themselves. It was a day of laying the groundwork for a journey that was only just beginning.

B. Day 2: Dockerization and Execution Woes

With the foundations set on Day 1, Day 2′s mission was to containerize the Rust environment using Docker to ensure a consistent development and execution environment. This was done with the aim of creating a lightweight and secure setup that could potentially be deployed anywhere with ease.

1. Docker Build: A Smooth Sail

The day began with a sense of optimism as I initiated the Docker build process. The Dockerfile was designed to use Alpine Linux as the base image, keeping the overall footprint small. The build sequence was crafted carefully to ensure that all Rust tooling and dependencies were in place:

The curl command was used to test the API connectivity within the container. The jq tool was installed to allow for command-line JSON processing. A new Rust project RustDataVoyager was created within the container. The project source code was copied into the container. A final cargo build command was invoked to compile the Rust application. The Docker image build log showed no errors, and the tag dataquest-rust-docker:latest was successfully applied to the image. It was a moment of accomplishment as the image was now ready to be instantiated into a container.

2. Execution Hurdles: The Binary Conundrum

However, my excitement was tempered when I attempted to run the newly created Rust application inside the Docker container. The command cargo run failed, with the error message indicating that the target/debug/rust_data_voyager binary could not be executed due to it not being found. This was perplexing as the binary should have been created during the build process.

I was dropped into an Alpine Linux shell prompt inside the container to further investigate the issue. Commands like uname -a confirmed the Linux environment, while a listing of the RustDataVoyager directory showed all the expected files in place, yet the binary remained elusive.

3. Troubleshooting: A Deep Dive

The troubleshooting process was meticulous:

I reviewed the build output during the Docker image build to ensure there were no hidden errors. I verified that the Dockerfile and build process were supposed to compile the Rust application and generate the binary. I checked the target/debug directory within the source code to look for the binary. I made sure that the Dockerfile set the working directory correctly to /usr/src/RustDataVoyager. Each step was taken with a careful approach to uncover the root cause of the missing binary. The process involved a lot of back-and-forth, examining build logs, and verifying directory paths.

4. *Reflections and Learnings*

Day 2 brought with it a stark reminder of the complexities of containerization and cross-environment development. While Docker promised a uniform environment, it also introduced new variables that required attention to detail and a deep understanding of both the Rust ecosystem and Docker's mechanisms.

Despite the setbacks, the day was rich with learning opportunities. It underscored the importance of verifying every step of the development pipeline, from writing code to deploying executables. It was a day that tested patience and perseverance but ultimately contributed to a growing repository of knowledge and skills.

C. Day 3: Rust Error Handling and Data Management

On Day 3, I confronted a critical juncture in my Rust learning journey. The initial idea of leveraging a live API for data science tasks was met with unexpected challenges. The API proved to be unreliable, or perhaps it was my inexperience with its intricacies that led to inconsistent results. In response, I pivoted my approach:

- Reliability: I opted for downloaded CSV files, which offered a more stable and reliable data source than the unpredictable API.
- Simplicity: This shift allowed me to focus my efforts on data processing with Rust, without the added complexity of API management.
- Control: Static CSV files provided a controlled environment for a more straightforward debugging and learning process.

Acknowledging the steep learning curve with the API, I redirected my project to work with CSV files, setting the stage for a more focused and fruitful exploration of data science with Rust.

1. *The Error Conundrum: Rust's Type System at Play*

The day's work began with an error handling issue that Rust's compiler brought to light. My attempt to create a file and write to it within a function that returns a Result<(), reqwest::Error> resulted in compilation errors. The compiler was unsparing in its feedback:

```rust
let mut file = File::create(&file_path)?;
file.write_all(&bytes)?;
```

The ? operator, which I had used to propagate errors up the call stack, was designed to work with compatible error types. However, the errors arising from file operations (std::io::Error) were not automatically convertible to the expected reqwest::Error. This was Rust's type system enforcing strict error type consistency.

I learned that I had to map the error to the appropriate type or change the function's return type to accommodate multiple error types using an enum or a boxed error type. This was a deep dive into Rust's powerful, yet sometimes complex, error handling model.

2. *Data Export Trials: CSV Format Handling*

The day also saw me exploring data exportation. I had been interacting with the KAPSARC API, trying to export datasets. After some trial and error, I decided to attempt exporting a dataset in CSV format. Using the curl command, I fetched the data and saved it to "main.csv". The command looked like this:

```
curl -X 'GET' 'https://datasource.kapsarc.org/api/explore/v2.1/catalog/datasets/the-renewable-
energy-policy-paradox/exports/csv' -H 'accept: application/json; charset=utf-8' -o main.csv
```

The CSV file, once opened, presented a new challenge — it contained metadata headers that needed parsing to access the actual dataset content:

```
recordid;_record_id;record_timestamp;_record_timestamp;...
```

I learned that dealing with real-world data often involves an additional layer of processing to separate useful information from metadata. This was an important lesson in data cleaning and preparation, essential skills for any data scientist.

3. *User Interaction and Support*

As part of the learning experience, I also engaged in discussions about how to handle the CSV data. The headers in the CSV seemed to pertain to metadata rather than the data itself. I advised on opening the file in a text editor or spreadsheet program to inspect the content and find the actual dataset data.

This interaction was a reminder of the importance of good communication skills in programming — being able to clearly explain and discuss issues is as crucial as solving them.

4. *Reflections on Progress*

The third day brought with it a recognition of the multifaceted nature of programming. It was not just about writing code; it was also about understanding the ecosystem in which the code operates and the data it manipulates. Error handling, data processing, and user support — each aspect was a piece of the larger puzzle.

The day concluded with a sense of accomplishment and a deeper appreciation for Rust's capability to enforce robustness through its strict type system. The journey through Rust was proving to be as much about learning the language as it was about understanding the principles of software development and data science.

D. Day 4: Transitioning Environments and Refining Data Handling

The fourth day of my data science journey with Rust was one of significant transitions and deep technical exploration. Embracing substantial changes to my development setup, I tackled the intricacies of data manipulation and began to reshape my project's structure for enhanced maintainability and scalability.

1. *Workflow Shift: From Docker to Local Development*

A pivotal shift occurred in my workflow as I moved away from Docker and Alpine Linux containers. I opted to run Rust locally on my Mac, which provided immediate benefits:

- Local Environment: The integrated development process and immediate feedback loop improved my efficiency and speed.
- Simplicity: I removed the overhead of Docker management, streamlining my focus onto Rust itself.
- Performance: By avoiding the potential limitations of containerized setups, my Mac's resources were better utilized.

This transition represented a growing understanding of my development tools and a commitment to optimizing the learning environment.

2. *Project Restructuring: Modularization for Clarity*

To address the increasing complexity of my project, I reorganized my codebase into several distinct modules:

- concatenation.rs: Handled the merging of disparate data sources.
- arrow_converter.rs: Encapsulated the logic for converting CSV data into the Apache Arrow format.
- data_analysis.rs: Contained functionalities for data analysis and insights extraction.
- data_validation.rs: Focused on ensuring data integrity and accuracy.

The introduction of a lib.rs file allowed these modules to be orchestrated cohesively, with main.rs now acting as the entry point that called upon the library for specific functionalities.

E. CSV Parsing: Overcoming API Limitations with Local Data

Faced with the challenges of an unresponsive API, I turned to downloading CSV files for local data analysis. The datasets included:

1. Table 2.4.4U. Price Indexes for Personal Consumption Expenditures by Type of Product.csv [1]
2. Table 2.4.5U. Personal Consumption Expenditures by Type of Product.csv [2]
3. Table 2.4.6U. Real Personal Consumption Expenditures by Type of Product, Chained Dollars.cv [3]

4. Table 2.6. Personal Income and Its Disposition, Monthly.csv [4]
5. Table 2.8.7. Percent Change From Preceding Period in Prices for Personal Consumption Expenditures by Major Type of Product, Monthly.csv [5]
6. Personal Consumption Expenditures Price Index.xlsx [6]

During the process, I encountered a schema mismatch error:

```
Error: InvalidArgumentError("number of columns(1) must match number of fields(47) in schema")
```

This led to the enhancement of the create_arrow_schema function to more accurately mirror the structure of the CSV files. Subsequently, I adopted the Apache Arrow format for its robust scalability and suitability for simulating complex data environments.

The decision to use Apache Arrow's columnar storage was pivotal in replicating real-world data workflows, reinforcing my commitment to utilizing appropriate tools for authentic data science endeavors.

F. Data Science with Rust: Evaluating Against Python

I invested time in evaluating Rust's data manipulation capabilities against Python's, tackling operations like CSV parsing, handling missing values, and adjusting data types. Rust provided granular control, contrasting with Python's user-friendly approach.

G. Insights on Language Proficiency and Prospective Pathways

Day 4 marked a significant milestone in my exploration of Rust, where I juxtaposed performance with usability. The insights gained not only deepened my understanding of Rust's strengths in data manipulation but also set the stage for future project hurdles.

H. Day 5: Overcoming Compilation Errors and Strategically Pivoting to Structured Data

Day 5 of my Rust journey was a blend of technical deep dives and strategic shifts. The day was split between navigating through compilation errors with Apache Arrow and making a pivotal transition to structured data for more efficient analysis.

1. Compilation Errors: A Steep Learning Curve

The initial part of the day was spent on addressing the complexities of Rust's type system and the Apache Arrow library, which were integral to my goal of validating and converting CSV data to the Apache Arrow format.

Type Mismatch Error: Faced with an `InvalidArgumentError`, I revised my code to dynamically align the Arrow schema with the CSV structure, as the number of columns in my CSV did not match the schema's fields.

Incorrect Method Calls: After encountering an error due to a non-existent method on the `FileReader` struct, I delved into the Arrow crate's documentation to correct my usage, which was a rich learning moment.

Incorrect Error Propagation: The misuse of the `?` operator led me to a better understanding of Rust's error handling patterns, refining my approach to iterators and non-`Result` types.

2. Embracing Apache Arrow for Scalability

Despite the initial setbacks, I remained committed to utilizing Apache Arrow, appreciating its scalability and the efficiency it offers for handling large datasets—key for real-world data science tasks.

3. Overcoming Data Complexity with Python Scripting

Acknowledging the complexity of the previous CSV files, I utilized a Python script to break down a comprehensive Excel file into multiple CSV files, leading to a cleaner and more manageable dataset.

4. The Shift to a Cleaned Dataset: 'organization-100.csv'

I pivoted to the 'organization-100.csv' dataset, which was not only cleaner but also free from the encoding issues of the 'iris.csv' file, thereby enhancing my ability to focus on data processing and analysis within Rust.

5. *Reflections on Adaptability and Tool Selection*

The day was marked by the importance of adaptability in data science. The strategic shift to a more suitable dataset, combined with the use of cross-language scripting for data preparation, underscored the iterative nature of programming and the adaptive approach necessary for managing data effectively.

Day 5 was both challenging and rewarding, a testament to the iterative nature of software development. Each error message provided a learning opportunity, and the strategic pivot to structured data emphasized the project's adaptability and growth.

I. Day 6: Refinement and Error Resolution in CSV to Arrow Conversion

Day 6 of my Rust for data science journey was dedicated to refining the CSV to Apache Arrow conversion process, with a focus on resolving compilation errors and improving data validation.

By Day 6, I was fully immersed in refining the CSV to Apache Arrow conversion process. With the project scope change on Day 3 already established, I could dedicate my efforts to overcoming the technical complexities of Rust and the Arrow library.

- Error Handling: I tackled type mismatches and data parsing errors head-on, each one offering new insights into Rust's strict type safety and error handling.
- Data Conversion: The conversion process from CSV to Apache Arrow format became smoother as I applied the lessons learned from previous days, appreciating the efficiency and scalability of the Arrow format for big data tasks.
- Technical Insight: Day 6 was marked by a series of breakthroughs in understanding and utilizing Rust's powerful features for data processing.

Reflecting on the decision made on Day 3, the benefits of working with CSV files were clear. It provided a more predictable and manageable environment for mastering Rust's data science tools, setting a strong foundation for future developments.

1. *Type Handling and Schema Validation*

I encountered a type mismatch error that prevented the successful compilation of my code. The issue was with the handling of an Arc object, a reference-counted pointer used in Rust to allow multiple ownership of data, which is especially useful for complex data structures like Apache Arrow schemas. The error message was clear:

```
Error: InvalidArgumentError("number of columns(1) must match number of fields(47) in schema")
```

This pointed to a mismatch between the expected schema and the CSV file's actual structure. To resolve this, I updated the validate_csv_input and convert_csv_to_arrow functions to ensure they handled the Arc correctly, providing a consistent and accurate schema for the data conversion process.

2. *Debugging Data Parsing Issues*

Another challenge was a ParseFloatError, indicating that the CSV data contained fields that could not be parsed into floating-point numbers as expected by the schema. This error highlighted the importance of robust data validation and error handling in data processing pipelines:

```
Error: ParseFloatError { kind: Invalid }
```

To address this, I had to carefully inspect the CSV file to ensure that all float fields were in the correct format and did not contain any invalid characters or formats. Adding more granular error handling in the validate_record function became necessary to gracefully handle such parsing errors and to provide detailed debugging information, such as the line number and field content that caused the issue.

3. *Progress and Reflections*

Despite these challenges, by the end of Day 6, I had made significant strides in my understanding of Rust's error handling and data validation mechanisms. I learned the importance of precise type handling when dealing with complex data structures and the necessity of thorough data validation to ensure the integrity of the conversion process.

Moreover, the day's work underscored the value of Apache Arrow in handling large datasets efficiently, validating my choice to use it for scalability and performance in data science tasks.

J. Day 7: Structuring for Documentation and Compliance

On the seventh day of my data science journey with Rust, I began with the administrative task of setting up a proper documentation and task management system. This was crucial as my project grew in complexity and required a more structured approach to keep track of the various components and dependencies involved.

1. Emphasizing Task Management

I started the day by creating a TODO.md file, which would act as a centralized checklist for my ongoing and upcoming tasks. This file is vital for maintaining a clear overview of my objectives and ensuring that nothing important slips through the cracks as the complexity of the project increases.

The primary item on my to-do list was to document my learning journey, a task that would not only solidify my understanding but also serve as a guide for future reference. This documentation was to be carried out using Typst, chosen for its combination of simplicity and power, ideal for a project where clarity is paramount.

```
# TODO
- [ ] Document my learning journey using Typst.
```

2. Commitment to Legal Acknowledgement

Recognizing the significance of the open-source crates that empowered my project, I decided to add another task aimed at creating a NOTICE.md file. This file would acknowledge the contributions of the various crates and their authors, as well as ensuring that I adhere to the licensing requirements associated with their use.

```
Copy code
- [ ] Add a `NOTICE.md` file for crediting all crates used in the project, with their respective
licenses.
```

3. Day's Reflection: Beyond Coding

Day 7 might not have been characterized by coding breakthroughs, but it was a day that highlighted the importance of the often overlooked aspects of software development: documentation and compliance. By addressing these areas, I was laying down a solid foundation for managing the project's growth and complexity.

The lessons from this day were clear—successful software development is multifaceted, with coding being just one aspect among many that include project management, documentation, and legal considerations. The groundwork laid on this day was crucial for a smooth journey ahead.

K. Day 8

on day 8 I added error_handling.rs for error handling, and throughout the day I was solving poblems, fixing bugs and errors I also added another file called read_csv.rs which reads and prints the csv file to the screen