Implementing Datadive: A Web Application for Data Analysis

Luca Schultz, 10. November 2024

Abstract

This thesis presents the design and implementation of Datadive, a web-based platform for data analysis. The platform aims to bridge the gap between GUI-based and code-based analysis tools by offering an intuitive interface while retaining the flexibility of programmatic approaches. At its core, Datadive features a cell-based interface, where each cell represents a step in the data analysis workflow. This design combines the accessibility of graphical interfaces with the power of code-based tools.

The thesis outlines the platform's architecture, which utilizes the Project Jupyter ecosystem for code execution and management, alongside a modern web stack for the backend and user interface. The system features a multi-tenant architecture with Kubernetes for scalability and isolation, incorporating separate databases for tenant and landlord functionalities. Architectural decisions emphasize maintainability and extensibility by using TypeScript throughout the stack and implementing robust error handling and dependency injection patterns.

While the current implementation serves as a technical foundation rather than a complete platform, it lays the groundwork for future development. The thesis concludes by outlining the next steps needed to transform this foundation into a fully functional data analysis platform. These steps include enhancing data management capabilities, adding user collaboration features, and developing statistical analysis assistance tools.

Table of Contents

1. Introduction
    1.1. Scope of this Thesis
2. Requirements
    2.1. Functional Requirements
    2.2. Technical Requirements
3. Architecture
    3.1. Project Jupyter Ecosystem
    3.2. Datadive Architecture
    3.3. Code Organization
    3.4. Data Model
4. Code Style
    4.1. Code Style Enforcement
    4.2. Tests
    4.3. Functional Programming
    4.4. Error Handling
    4.5. Dependency Injection
    4.6. Factory Functions
    4.7. Enum Like Objects
5. Technologies
    5.1. Typescript
    5.2. Bun
    5.3. Kubernetes
    5.4. Hono.js & Related Libraries
    5.5. libSQL
    5.6. Kysely & Kysely Codegen
    5.7. React
    5.8. TanStack Router and TanStack Query

- 5.9. Vite & Vitest
- 5.10. Tailwind CSS
- 5.11. Astro.js
- 5.12. Zod
- 5.13. neverthrow
- 5.14. Radix UI
- 5.15. ESLint
- 5.16. Prettier
- 5.17. Turborepo

6. Third Party Services
   - 6.1. Turso
   - 6.2. Resend
7. Packages
   - 7.1. The @datadive/db Package
   - 7.2. The @datadive/email Package
   - 7.3. The @datadive/utils Package
   - 7.4. The @datadive/auth Package
   - 7.5. The @datadive/spec Package
   - 7.6. The @datadive/core Package
   - 7.7. The @datadive/jupyter Package
   - 7.8. The Configuration Packages
8. Applications
   - 8.1. The @datadive/api App
   - 8.2. The @datadive/web App
   - 8.3. The @datadive/docs App
9. Setup Development Environment
   - 9.1. Jupyter Hub
   - 9.2. VSCode
   - 9.3. Bun
   - 9.4. Turso Account

- 9.5. Resend Account
- 9.6. Repository Setup
  - 9.6.1. Environment Files
- 9.7. Running Datadive
- 9.8. Creating a Tenant

10. Outlook
- 10.1. Challenges and Limitations
- 10.2. Next Steps
- 10.3. Beyond the Core Functionality
- 10.4. Conclusion

1. Introduction

Users of data analysis software are often not statisticians or data scientists with extensive programming skills. Instead, they are typically experts or students in their own research fields who have learned established statistical methods to apply to datasets with known structures. Since graphical user interfaces (GUIs) are the most common way to interact with software, it is no surprise that these users often rely on GUI-based applications for their data analysis tasks. [1]

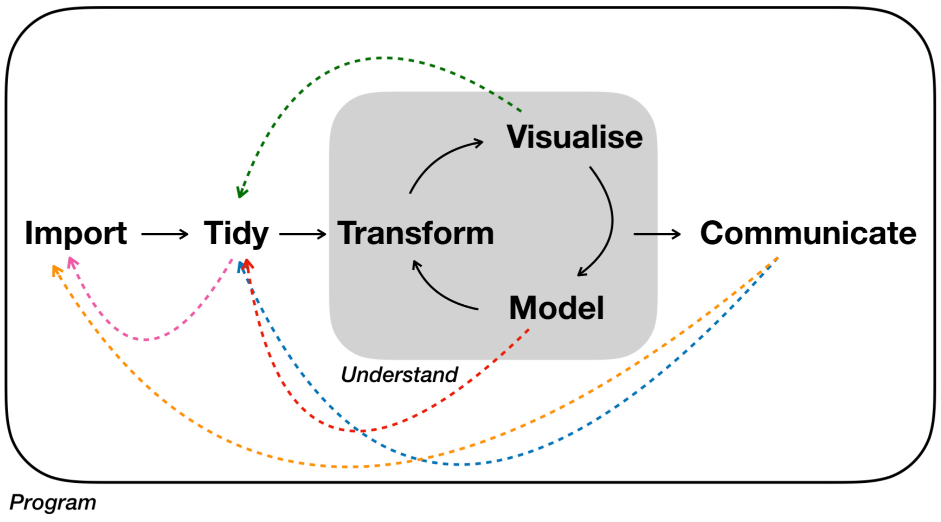

Fig. 1: Diagram of a realistic data analysis workflow. The black arrows represent the idealized workflow, while the dashed arrows illustrate the actual workflow. The diagram is taken from the online book "Robust data analysis: an introduction to R" by Sina Rüeger. [2]

While GUI-based data analysis tools offer a familiar interface for users without programming experience, they often have limitations that can impede the analysis process. Data analysis is complex and nonlinear, typically requiring multiple iterations of trial and error, as illustrated in figure 1. GUI-based tools often constrain users to a predefined workflow, which can be restrictive. If a specific functionality is unavailable in the GUI, users must export and re-import the data to use another tool to complete a step in the analysis. Additionally, reproducing an analysis is challenging because the steps taken are not recorded in a script that can be rerun. [3] [2]

Recognizing the challenges of current data analysis approaches, the Behavioral Security Research Group at the University of Bonn is developing a data analysis tool as part of Florin Martius' doctoral thesis: Datadive. This tool aims to provide a GUI-based platform that encompasses the entire workflow of scientific data analysis projects. During the conceptual phase, discussions with Florin Martius and other research group members identified the platform's requirements. Based on these requirements, a concept for the platform was developed.

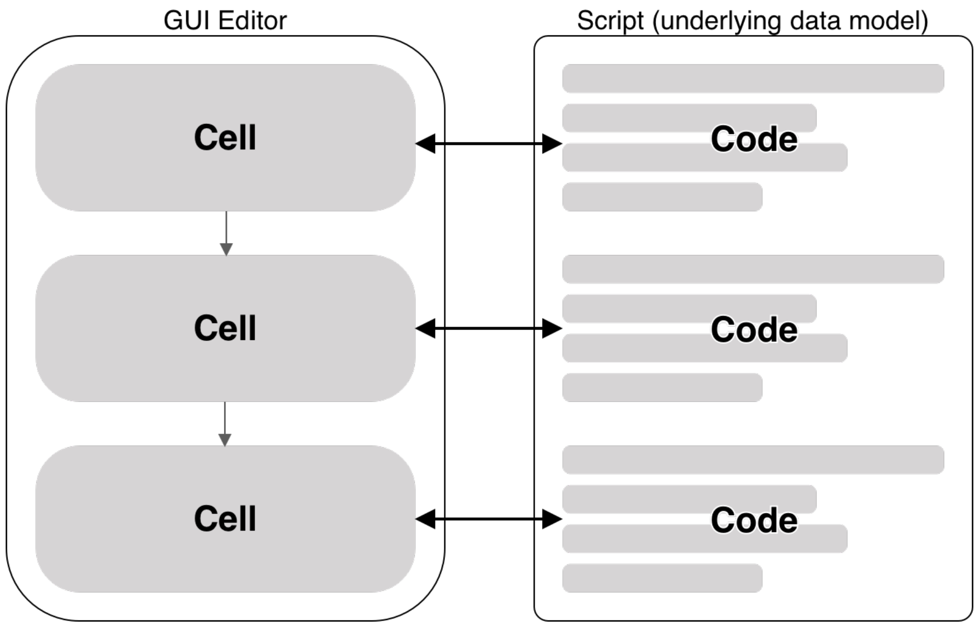

Fig. 2: A simplified representation of the Datadive data model. Each cell corresponds to a piece of code. All cells together form a script that contains the analysis workflow.

The central component of Datadive is a cell-based interface, where each cell represents a step of the data analysis workflow. Each cell receives the output of the preceding cell and additional parameters provided by the user as input. The cells are arranged in a sequence that represents the entire analysis workflow. Users can conduct the analysis by adding, deleting, or reordering cells. To revisit a previous step, users can replace a cell or modify its input parameters.

A simplified description of the underlying data model, illustrated in figure 2, is that each cell corresponds a piece of code. All cells together form a script that users can manually edit if needed. This approach combines the flexibility of code-based tools with the user-friendliness of GUI-based tools. The platform is accessible to users without programming expertise while remaining flexible enough for expert users to write code when certain functionalities are unavailable in the GUI. Additionally, the data model ensures reproducibility by recording the steps of the analysis in a script that can be rerun.

1.1. Scope of this Thesis

The conceptual part of the thesis yielded requirements and the already described concept for the Datadive platform.

During the practical part, an architecture that implements this concept was developed, and a technical foundation for it was established. The resulting codebase is not a functional platform but serves as a starting point for future development. The main focus was on creating a maintainable codebase. This is particularly important as Datadive is developed within the context of a doctoral thesis and will be a collaborative project between the research group staff and students at the University of Bonn. To facilitate collaboration, great emphasis was placed on making it easy for staff to ensure contributions meet the project's quality standards and for developers to start contributing. Complex solutions were avoided, and the number of technologies used was limited to reduce the learning curve for new developers.

The written portion of this thesis outlines the requirements established during the conceptual phase. It describes the developed architecture and details the structure and organization of the codebase, serving as documentation. Additionally, it references the technical decisions made during development and provides an outlook on the future work needed to make the Datadive platform fully functional.

This documentation is also available as a hosted website, featuring slightly different content. The structure of both formats is loosely based on the Diataxis framework for technical documentation proposed by Daniele Procida. Both formats contain an *explanation* section that provides context and connections to the Datadive platform, a *reference* section that offers concise information about specific elements or features and a *guide* section that provides step-by-step instructions for contributors. [4] [5]

2. Requirements

This thesis distinguishes between functional and technical requirements, as they serve different purposes in software development and must be managed differently throughout the project lifecycle.

2.1. Functional Requirements

The functional requirements for the Datadive platform were derived from the needs of users. The key requirements, in no particular order, are:

- *Comprehensiveness*: In the future, the platform should encompass the entire workflow of scientific data analysis projects, including data import, tidying, transformation, analysis, and visualization.
- *Flexibility*: The platform should allow users to write code instead of relying solely on the built in functionalities. This ensures the platform is adaptable to the specific needs of researchers and can be used even if not all features are available in the GUI.
- *Extensibility*: The platform should be extensible, allowing users to customize their workflows by adding new functionalities. These extensions should be reusable across users and projects, enabling users to permanently enhance the platform's capabilities.
- *Integration*: The platform should integrate with external tools and services commonly used in scientific research. It should support importing and exporting data in commonly used formats, such as CSV, JSON and common Excel formats.
- *Collaboration*: The platform should support collaboration between users by allowing them to share projects, notebooks, and files.
- *Helpful Feedback*: The platform should provide users with helpful feedback and guidance on how to use its features effectively. For example, an integration with the Qualtrics survey maker could allow users to import Qualtrics survey files

and use the included metadata to recommend appropriate statistical tests for analyzing the data.

- *User-Friendly Interface*: The platform should have a user-friendly interface that is easy to navigate and understand.
- *Security & Privacy*: The platform should ensure the security and privacy of user data by implementing appropriate access controls and security mechanisms. It should collect only the necessary data and provide users with control over their data.
- *Accessibility*: The platform should be accessible to users with disabilities.

2.2. Technical Requirements

These requirements are focused on the technical aspects of the platform and guide the development of the Datadive platform. The key technical requirements are:

- *Maintainability*: The platform should be maintainable, allowing developers to easily understand and contribute to the codebase while working independently on different parts of the platform.
- *Isolated Environments*: The platform should provide isolated environments for users to work in, ensuring that each user has their own workspace to store data and execute code. This prevents interference between users, ensures data privacy and security and prevents the execution of malicious code on Datadive's servers.
- *Arbitrary Code Execution*: To provide fallback options for users whose requirements are not not met by the built-in functionality , the platform should support writing and executing arbitrary code.
- *Customizable Execution Environments*: The platform should allow users to customize their execution environments by installing additional libraries and packages which are not part of the default environment.
- *Performance*: The platform should be performant, providing users with a responsive and smooth experience. This includes fast loading times, quick execution of code, and minimal latency in interactions.

- *Reliability*: The platform should be reliable, ensuring that users can access their data and work without interruptions. This includes high availability, fault tolerance, and data durability.
- *Scalability*: The platform should be scalable to support a large number of users and data analysis tasks. This ensures that the platform can handle increasing workloads and user demands without performance degradation.

These requirements guided the architectural decisions made during this thesis, ensuring the platform is well-positioned to meet these requirements as it evolves.

3. Architecture

Software architecture defines the structure of a software system, detailing its organization and the interaction of its components. It serves as a blueprint to ensure the system is scalable, reliable, and maintainable. Effective software architecture is crucial for managing complexity and guiding development to meet requirements. This thesis distinguishes between two architectural layers: code organization and system architecture. Code organization refers to the structure of the codebase, while system architecture addresses the high-level arrangement of components (e.g., the backend, a microservice, or a client app) and their interactions during runtime[1]. [6]

This chapter provides an overview of the Datadive platform's architecture, explaining the decisions and rationale behind it. It describes the high-level architecture independently from the technologies used. The following chapters discuss the selected technologies, the reasons for their choice, and the resulting code organization.

3.1. Project Jupyter Ecosystem

The key requirement that shaped the architecture of the Datadive platform was the ability for users to write code rather than depend solely on the GUI for data analysis. This choice aimed to increase the platform's flexibility, enabling it to meet researchers' specific needs and remain functional even when not all features are accessible through the GUI. Additionally, this decision resulted in an architecture based on Project Jupyter, which provides components for authoring and executing code in a web environment. This section outlines Project Jupyter, focusing on the components relevant for the Datadive platform and their interactions.

Project Jupyter offers a platform for writing code, visualizing data, and sharing results. The code is stored in notebooks, which are organized into cells that can contain code, text, or visualizations. This cell-based interface allows users to write and execute code interactively. Project Jupyter offers several components that can be combined in various ways for use either locally or in a server environment. Project Jupyter is used by institutions like the University of Berkeley, the University of Sheffield or the Michigan State University and companies such as Bloomberg, Google and IBM. In 2017, a Team at UC San Diego analyzed over 1 million notebooks stored in public GitHub repositories [7]. In 2019 a team of project Jupyter contributors found nearly 5 million notebooks stored in public repositories in a similar effort, both numbers suggesting that Project Jupyter is a widely used tool for data science tasks. [8] [9]

```
{
  "metadata": { ... },
  "cells": [
    {
      "cell_type": "markdown",
      "metadata": { ... },
      "source": ["some *markdown*"]
    },
    {
      "cell_type": "code",
      "execution_count": 1,
      "metadata": { ... },
      "source": ["print('hello, world!')"],
      "outputs": [ ... ]
    }
  ]
}
```

Fig. 3: Simplified example of the Jupyter Notebook JSON format. [10]

When writing code in Jupyter, users interact with either Jupyter Notebook or JupyterLab. Both are JavaScript applications that run in web browsers. Jupyter Notebook is the original interface, while JupyterLab is the next-generation interface that provides enhanced functionality. JupyterLab allows users to open multiple notebooks or files, such as HTML, text, and Markdown, as tabs in the same window. It also offers a user experience similar to that of an integrated development environment[2] (IDE). Both interfaces enable users to write and execute code in a notebook format. [9] [11]

As illustrated in figure 3, the code is stored in notebooks, which are JSON files. The top level object of the JSON file contains metadata about the notebook, such as the kernel used to execute the code. The `cells` array contains the individual cells of the notebook, which can be either code cells or markdown cells. Code cells contain the source code to be executed, while markdown cells contain text formatted using Markdown. The notebook format allows users to write code, visualize data, and share results in a single document. [10]

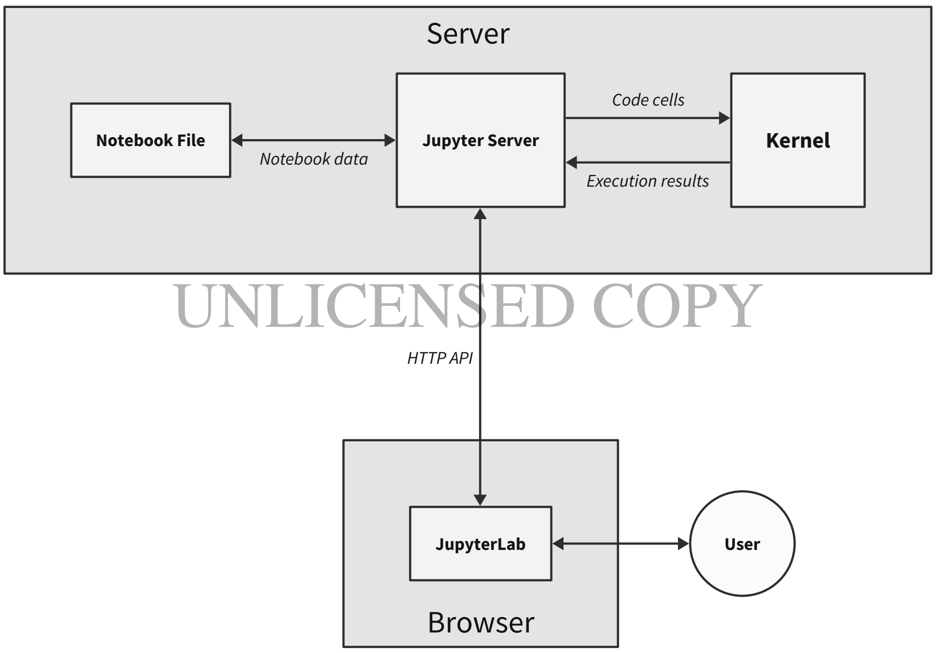

Fig. 4: Simplified overview of the components required for Jupyter code execution [11].

Kernels are used to execute code in various programming languages. A kernel communicates through a lightweight messaging protocol called ZeroMQ. It executes the code sent to it and responds with the results. Additionally, it offers code completion and maintains the state of the code execution during sessions. The Jupyter ecosystem provides kernels for several programming languages, including Python, R, and Julia. [11] [12]

The Jupyter Server acts as the communication hub between these components, as shown in figure 4. It is responsible for saving and loading notebooks, processing user interactions (e.g., executing code cells), and managing the kernels. User interactions occur through an HTTP[3] API[4], which the Jupyter Notebook and JupyterLab interfaces use to communicate with the Jupyter Server. [11]

Jupyter Servers do not have a concept of users or access control. Anyone who can access a jupyter server can also access all notebooks and execute arbitrary code. To address this, the Jupyter Servers can be managed by JupyterHub, which is a multi-user server that provides access to Jupyter Servers for groups of users. It manages user authentication, access control and resource allocation like starting and stopping Jupyter Servers for users. [11]

3.2. Datadive Architecture

As illustrated in figure 5, the architecture of the Datadive platform comprises three main components: the frontend, the backend, and the Jupyter components. Jupyter Hub manages the starting and stopping of Jupyter servers for each user within a Kubernetes cluster. Kubernetes is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. The Jupyter servers are based on Docker[5] images and run in Docker containers. They handle notebooks, related files, and code execution. Using separate Jupyter servers for each user ensures isolated environments for file storage and code execution. Additionally, since Jupyter Hub supports customizing the Docker images used to run the Jupyter servers, future versions of Datadive may allow users to install additional libraries and packages in their execution environments [13].

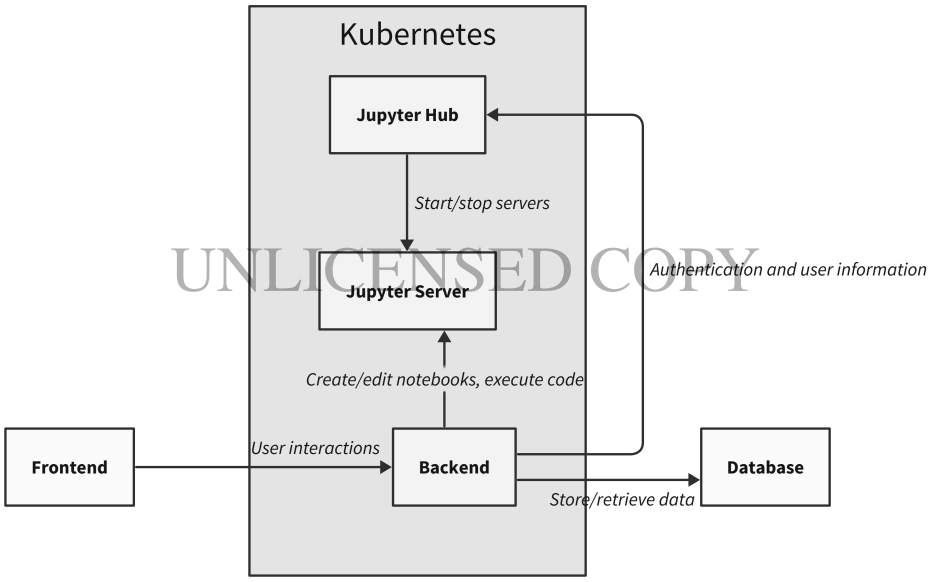

Fig. 5: Overview of the Datadive platform architecture.

The backend communicates with the Jupyter components through their respective HTTP APIs [14] [15] [16]. It acts as a communication hub between the frontend and the Jupyter components, but also provides additional functionality, such as user management, and create, read, update and delete[6] (CRUD) operations for projects, notebooks and other resources. It abstracts the complexity of the Jupyter components and provides a simplified interface for the frontend to communicate with. The frontend is a single-page application[7] (SPA) that communicates with the backend using a HTTP API.

This architecture allows for a clear separation of concerns between the frontend, backend, and Jupyter components. The frontend is responsible for rendering the user interface and handling user interactions, while the backend manages the business logic and communicates with the Jupyter. The Jupyter components handle the execution of code and the management of notebooks.

In addition to separating concerns, the decision to divide the application into frontend and backend components and use a HTTP API for communication aimed to enable independent development and maintenance. Both the backend and frontend can be built according to an API specification, which outlines the endpoints and data structures for communication. This specification then serves as a contract between the frontend and backend and is used for generating an API client in the frontend and for inferring endpoint return types in the backend, enforcing the contract between the two components.

This strategy introduces some additional overhead and may slow down development. However, it enables different teams to work on the frontend and backend simultaneously without interfering with each other's tasks. As long as the API specification is followed, changes can be made to either the backend or frontend without affecting the other component. If a feature requires modifications to the API, those changes are reflected in the API specification, allowing both the frontend and backend to be updated independently. To facilitate development, the API specification can be used to mock[8] the backend API, enabling frontend developers to work without a fully functional backend. Meanwhile, backend developers can utilize API clients like Postman or cURL to make requests to the backend and develop the API independently of the frontend.

3.3. Code Organization

Datadive is a complex software system composed of multiple components, each with specific responsibilities and interactions. To manage this complexity, the codebase is organized into separate modules, each focusing on a particular aspect of the system. This chapter provides an overview of Datadive's code organization, explaining the structure of the codebase and the rationale behind it.

Datadive employs a monorepo structure, where all the code for its frontend, backend, and Jupyter components is contained within a single repository. This approach provides several benefits. Firstly, it simplifies development by allowing developers to work on different parts of the system without the need to switch between multiple repositories, facilitating the implementation of features that span multiple components. It also ensures consistent versioning, as all components are updated together, preventing compatibility issues that might arise from changes to individual components. Furthermore, shared configurations, such as configuration files and build scripts, are centrally stored, which reduces duplication and maintains uniformity across the system. The monorepo setup supports atomic commits, enabling changes affecting multiple components to be committed simultaneously, thereby simplifying the review and merging processes. Additionally, refactoring code that spans multiple components is more straightforward when all the code resides in a single repository. Lastly, code sharing is enhanced, as utilities, configurations, and specifications can be easily accessed and used across different components, promoting efficiency and consistency. [17]

```
datadive
├── CODE_OF_CONDUCT.md      # Code of conduct
├── CONTRIBUTING.md         # Contribution guidelines
├── LICENSE                 # Project license
├── README.md               # Project README
├── SECURITY.md             # Security guidelines
├── apps/
│   ├── api/                # @datadive/api package
│   ├── docs/               # @datadive/docs package
│   └── web/                # @datadive/web package
├── bun.lockb*              # Bun lockfile
├── cspell.config.yaml      # CSpell configuration
├── eslint.config.js        # Root ESLint configuration
├── package.json            # Root package.json
├── packages/
│   ├── auth/               # @datadive/auth package
│   ├── config/
│   │   ├── eslint/         # @datadive/eslint package
│   │   └── tsconfig/       # @datadive/tsconfig package
│   ├── core/               # @datadive/core package
│   ├── db/                 # @datadive/db package
│   ├── email/              # @datadive/email package
│   ├── jupyter/            # @datadive/jupyter package
│   ├── spec/               # @datadive/spec package
│   ├── turso/              # @datadive/turso package
│   ├── ui/                 # @datadive/ui package
│   └── utils/              # @datadive/utils package
├── patches/                # Patches for dependencies
├── prettier.config.js      # Prettier configuration
├── scripts/                # Scripts for common tasks
├── thesis/                 # Written thesis
├── tsconfig.json           # TypeScript configuration
└── turbo.json              # Turborepo configuration
```

Fig. 6: Overview of the Datadive monorepo structure, it's packages and configuration files.

The Datadive monorepo is managed by Bun workspaces, which allow the codebase to be divided into several packages stored within the monorepo. In this context, packages are reusable pieces of code that can be installed and integrated into a software development project as dependencies. These packages are then combined to build the Datadive components. [18]

As shown in figure 6, the repository's structure includes two main directories at the root of the project: `/apps` and `/packages`. These paths are relative to the root of the repository, containing the version control data for Datadive. The `/apps` directory includes the code for the frontend, backend, and documentation applications. In contrast, the `/packages` directory holds packages used by other packages or applications.

Datadive organizes code into packages to promote reuse and enforce architectural boundaries. Each package encapsulates specific functionality, such as database access, authentication logic, or business logic. This approach allows contributors to focus their development efforts on individual packages without feeling overwhelmed by the entire codebase. A subcategory of packages is dedicated to configuration and is stored in `packages/config`. These packages contain shared configuration files, such as ESLint or TypeScript configurations.

Each package has a consistent structure, as shown in figure 7. Typically, all configuration files are located at the root of the package, where the `package.json` file is stored. This file contains machine-readable information about the package, such as its name, version, and the entry point for accessing the code. The following paths are relative to the package root..If the package generates build output, it will be found in the top-level `dist` directory. Scripts for common development tasks are stored in the `scripts` directory. The source code is located in the `src` directory, organized into subdirectories that often separate tenant-specific and landlord-specific code. Packages often include an `src/errors` directory that contains all the errors the package can produce. The `src/index.ts` file exports the package's public API, serving as the entry point for other packages to import its functionality.

```
package/
├── dist/                           ← Build output
├── scripts/                        ← Scripts for common tasks
├── src/                            ← Source code of the package
│   ├── landlord/                   ← Landlord-specific code
│   │   ├── tenant/                 ← Tenant management feature
│   │   │   ├── create-tenant.ts
│   │   │   └── shared/             ← Shared code for tenant management
│   │   └── user/                   ← User management feature
│   │       ├── list-users.ts
│   │       └── update-user.ts
│   ├── shared/                     ← Shared code for landlord features
│   │   ├── types/                  ← Shared landlord types
│   │   │   └── user.ts
│   │   └── utils/                  ← Shared landlord utilities
│   │       ├── parse-cookie.ts
│   │       └── parse-date.ts
│   └── tenant/                     ← Tenant-specific code
│       ├── notebook/               ← Notebook feature
│       │   ├── execute/            ← Notebook execution feature
│       │   └── update-notebook.ts
│       ├── shared/                 ← Shared code for tenant features
│       │   └── types/              ← Shared tenant types
│       │       └── notebook.ts
│       └── user/                   ← User feature
│           ├── update-user.ts
│           ├── delete-user.ts
│           └── shared/             ← Shared code for user feature
│               └── utils/          ← Shared user utilities
├── package.json
┆
```

Fig. 7: Example structure of a package in the Datadive codebase.

Within the `src` or `src/{tenant|landlord}` directory, the code is organized by feature or functionality. All code related to a specific feature is stored in a single file or, if extensive, in a directory called a "feature directory" in this thesis. Feature directories can contain other feature directories and represent a distinct scope of functionality, which narrows as the directory depth increases. For example, the core package may include a feature directory for user-related functionality at `src/tenant/user` and another for notebook-related functionality at `src/tenant/notebook`. Within the latter, the `src/tenant/notebook/execution` directory has a more specific scope, containing code related to notebook execution.

Each feature directory may include a `shared` directory that contains shared code used within the feature or across sub-features. This directory is typically organized by concern, with subdirectories for shared constants, utilities, or types. According to convention, shared code should reside in the lowest shared directory or the narrowest scope possible, meaning it should be as close as possible to the code that uses it. For example, if a utility function is used in more than one file but specific to one aspect of a feature, such as notebook execution, it should be placed in the `src/tenant/notebook/execution/shared` directory. If the utility function is used in multiple aspects of the feature, it should be in the `src/tenant/notebook/shared` directory. If it is utilized across multiple tenant specific features, it should be located in the `src/tenant/shared` directory.

The amount of nesting within feature directories should be kept as low as possible without storing an excessive number of source code files in a single directory. This thesis cannot provide an objective metric for when a feature directory should be divided into multiple subdirectories. While some numbers seem obviously incorrect—such as creating subdirectories for single files or keeping a thousand files in one directory—it is unrealistic to establish a simple set of rules that apply to all cases. Instead, future maintainers of Datadive will need to make this decision on a case-by-case basis as the codebase grows and evolves.

File names are chosen to reflect the content and purpose of the file. Files that export a single function or class are named after the exported entity. In contrast, files that export multiple entities are named either after the primary entity of the functionality they contain. For example, a file exporting a single utility function might be named `get-notebook-path.ts`, while a file exporting multiple utility functions could be named `notebook-utils.ts`. Typically, file names are written in kebab case[9] to support case-sensitive file systems and enhance readability.

This structure follows the principle of Locality of Behavior (LoB), which asserts that the behavior of a unit of code should be clear by examining only that unit. While the principle of locality of behavior often conflicts with the Don't Repeat Yourself (DRY) principle and the Separation of Concerns (SoC) principle, no research indicates that any of these principles is more important for codebase maintainability than the others. However, some prominent computer scientists suggest that locality of behavior may be the most crucial principle for code maintainability. For example, in his book *Patterns of Software*, Richard P. Gabriel emphasizes that the key feature for easy maintenance is locality. The decision to organize much of Datadive's code around the principle of locality of behavior reflects personal preferences, as do many other choices in software development. Nevertheless, locality of behavior is a well-established principle in software development, centered around the idea of making code easy to understand. This is particularly important for a project like Datadive, where contributors may frequently change due to the nature of a project that is developed in collaboration between students and researchers.

3.4. Data Model

The data model is a critical part of Datadive's architecture. It defines how data is organized, stored, and accessed within the system. The model provides an abstract representation of the application's data structures and relationships. This chapter outlines the data model used developed for Datadive and explains how it ties into the desired functionality of the platform.

Since Datadive relies on Jupyter components for code execution and management, the data model is aligned with Jupyter's data structures. This alignment simplifies the integration of Jupyter components and leverages the familiarity of users accustomed to Jupyter's terminology and concepts to enhance usability. At the core of the data model are users, projects, notebooks, and cells.

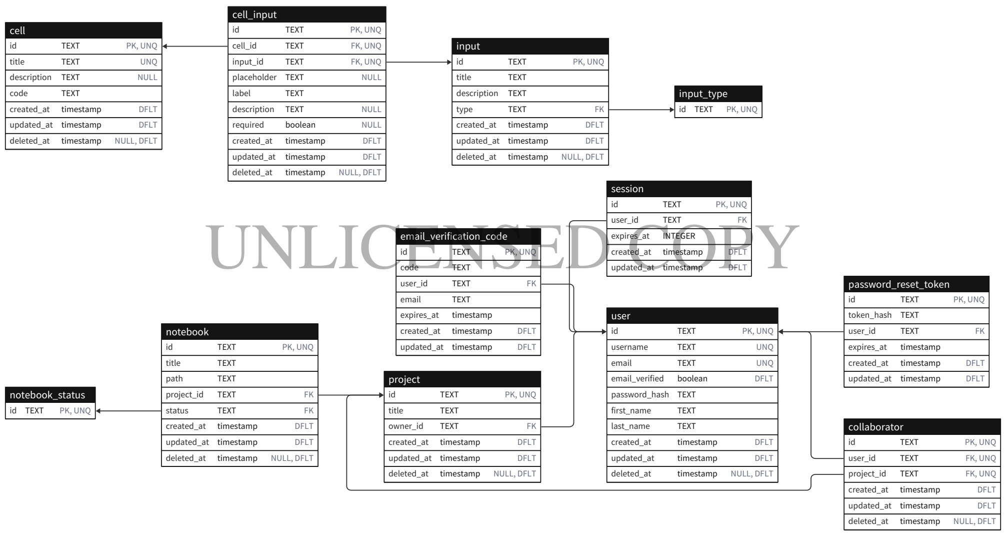

Fig. 8: The database schema of the tenant database as ER diagram.

Each Datadive user corresponds to a user in the data model, as shown in figure 8. Users create and own projects, which store metadata such as the project name and connection details to a Jupyter server instance created using JupyterHub. Each project contains notebooks, but Datadive only stores notebook metadata in the database, including the notebook name and associated project. The Jupyter server instance manages the notebook content: cells, the basic units of code, text, or other content.

Jupyter supports several cell types, with code cells and display data cells being most relevant for Datadive. Code cells contain executable code, while display data cells contain the output of code execution, such as text, images, or plots. The key difference from using Jupyter notebooks directly is that Datadive users do not create cells by writing code. Instead, they select from predefined cell templates which contain code snippets for data analysis tasks, such as loading data from a file, cleaning data, or performing statistical analysis. The code can contain placeholders for user input provided through the Datadive user interface. Each placeholder has an associated input with a name and a type, which can be a string, number, path, or a variety of other value types. These inputs generate the user interface for the cell, enabling users to provide the necessary information for the code snippet.

Many of the key interactions in Datadive revolve around cells. Users can create, read, update, and delete cells within a notebook. When a user executes a cell, Datadive requests the Jupyter server execute the notebook. The server processes the code and returns the output, which Datadive displays to the user. In the future, Datadive will support more advanced interactions, such as the creation of custom cell templates, the integration with external services or plugins to provide additional functionality, and storing the execution history of each cell. Another important part of the initially planned features are interactive cell templates. These could be used to guide users through complex data analysis tasks such as test selection or data cleaning.

One intentionally simplified part of the core data model in the initial implementation is dataset storage. Since Jupyter supports file uploads and notebook code can access the file system, Datadive does not store datasets in the database. Instead, users upload datasets through the Datadive HTTP API to the Jupyter Server, where they can be accessed in code cells. This approach simplifies the data model and reduces the complexity of managing large dataset storage in the database or a similar system. However, it also limits the platform's capabilities, as users cannot easily share datasets between projects or access them through the Datadive user interface if the underlying Jupyter server is not running. Future work on Datadive will need to improve dataset management, including the ability to upload, share, and visualize datasets directly in the platform.

Apart from the core data model, Datadive also offers features like user management, authentication, project sharing, and collaboration. These features are implemented using additional data structures and relationships that extend the core model. For example, authentication involves data structures for validating emails, resetting passwords and managing user sessions. Future versions of Datadive may also include more advanced user management features, such as roles and permissions to control access to projects and notebooks.

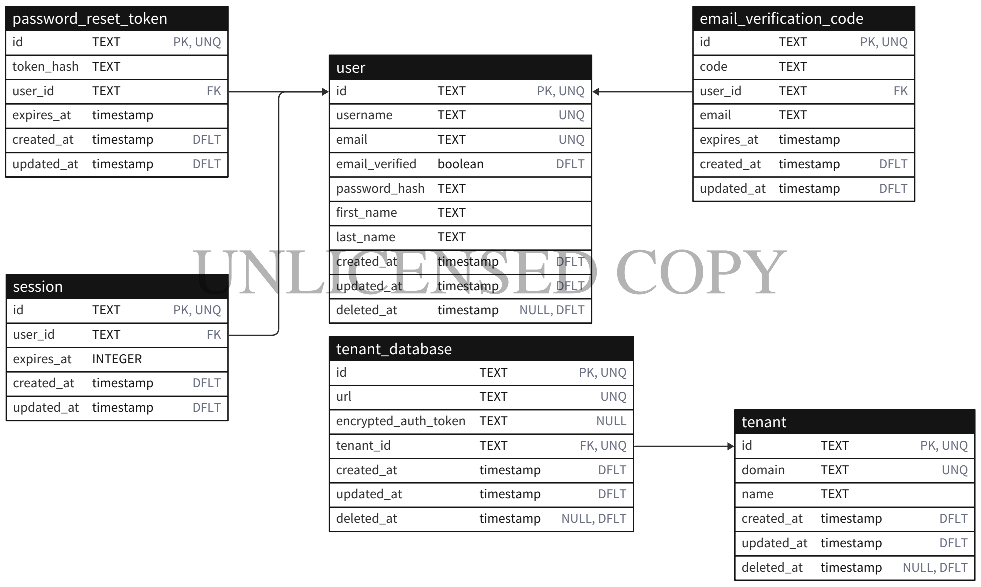
password_reset_token
id TEXT PK, UNQ
token_hash TEXT
user_id TEXT FK
expires_at timestamp
created_at timestamp DFLT
updated_at timestamp DFLT
user
id TEXT PK, UNQ
username TEXT UNQ
email TEXT UNQ
email_verified boolean DFLT
password_hash TEXT
first_name TEXT
last_name TEXT
created_at timestamp DFLT
updated_at timestamp DFLT
deleted_at timestamp NULL, DFLT
email_verification_code
id TEXT PK, UNQ
code TEXT
user_id TEXT FK
email TEXT
expires_at timestamp
created_at timestamp DFLT
updated_at timestamp DFLT
session
id TEXT PK, UNQ
user_id TEXT FK
expires_at INTEGER
created_at timestamp DFLT
updated_at timestamp DFLT
tenant_database
id TEXT PK, UNQ
url TEXT UNQ
encrypted_auth_token TEXT NULL
tenant_id TEXT FK, UNQ
created_at timestamp DFLT
updated_at timestamp DFLT
deleted_at timestamp NULL, DFLT
tenant
id TEXT PK, UNQ
domain TEXT UNQ
name TEXT
created_at timestamp DFLT
updated_at timestamp DFLT
deleted_at timestamp NULL, DFLT

Fig. 9: The database schema of the landlord database as ER diagram.

The most complex supportive feature in the initial implementation is multi-tenancy. Multi-tenancy is a software architecture where a single instance of an application serves multiple customers, known as tenants. Tenants are typically organizations with many users who need to manage their users and data, including custom branding, tenant-specific features, plugins, and configurations. In Datadive, multi-tenancy is implemented through a separate landlord application that creates and manages tenants while routing requests to the appropriate tenant database. Each tenant has its own database, ensuring that their data is isolated and inaccessible to other tenants. This architecture features two distinct data models: one for tenant data and another for landlord data. The landlord data model, shown in figure 9, is much simpler. It only needs to store information about tenants, connection details for their databases, landlord users, and authentication details.

4. Code Style

The previous chapter discussed the architecture of the Datadive platform, which refers to the high-level structure of a software system. It focused on how components interact and the design choices, such as microservices versus monolithic systems. Within this framework, code style plays a crucial role by establishing conventions and guidelines for writing code, including formatting and naming conventions. While architecture shapes the overall performance and reliability of the software, a consistent code style ensures clarity and coherence at the code level, enhancing readability and maintainability within a project. Companies or large projects often have a code style guide to ensure that all contributors follow the same conventions, two notable examples are Google and Deno but there also many recommendations which are based on the code style of open source code bases. [19] [20] [21]

This chapter introduces the code style used in the Datadive platform, outlining conventions and guidelines. It covers various aspects of code style, such as formatting, naming, documentation, and testing. By adhering to these guidelines, Datadive contributors can write code that is consistent, readable, and maintainable, facilitating collaboration and ensuring the quality of the software. The chapter starts with a brief discussion of the tools and processes that support code style enforcement in the Datadive codebase.

4.1. Code Style Enforcement

Although it is recommended that the process of contributing to Datadive includes a code review by a maintainer, it is unlikely that all maintainers will be able to catch every style violation. Therefore, maintaining a consistent code style across a project requires tools that automate the enforcement of style guidelines. These tools help ensure that all contributors follow the same conventions, reducing the likelihood of inconsistencies and errors. The Datadive platform uses several tools to support code style enforcement, the most important being ESLint and Prettier. ESLint is a static code analysis tool that identifies problematic patterns in JavaScript and TypeScript code. In Datadive, it also enforces a consistent code style. Datadive includes several ESLint configurations stored in the `@datadive/eslint` package, based on a `base` configuration described in more detail below. The other configurations mainly consist of framework-specific rules for React and Playwright, which will not be discussed in this chapter.

PLUGIN	PURPOSE	LINK
`eslint-plugin-jsdoc`	Enforces consistent JSDoc comments for functions and variables	github.com/gajus/eslint-plugin-jsdoc
`eslint-plugin-security`	Identifies security vulnerabilities in the codebase	github.com/eslint-community/eslint-plugin-security
`eslint-plugin-eslint-comments`	Enforces consistent comments in the codebase	github.com/mysticatea/eslint-plugin-eslint-comments
`eslint-plugin-turbo`	Includes rules for managing monorepos	github.com/vercel/turborepo

Table 1: ESLint plugins used in the Datadive codebase.

Most stylistic ESLint rules used in Datadive are based on the `typescript-eslint` ESLint plugin, which extends ESLint to support TypeScript-specific rules. It offers three shared configurations that serve as presets for sets of rules. Datadive uses the recommended, strict, and stylistic configurations, each in their type-checked version. The rules are detailed in the plugin's documentation. Notably, Datadive prefers using TypeScript interfaces over types, as this can improve type-checking performance. It also uses the `Array<>` syntax instead of `[]` for array declarations, which generally enhances readability and maintains consistency with other generic types. Additionally, it favors the index signature syntax `{ [key: string]: string }` over the `Record<string, string>` syntax because it allows for defining recursive types and is closer to the mapped type syntax `{ [K in keyof T]: U }`, making it less confusing for contributors when first encountered. In addition to the `typescript-eslint` plugin, Datadive uses the recommended shared configurations of several other ESLint plugin, which are detailed in table 1. [22] [23] [24] [25]

Prettier is a code formatter that automatically applies a predefined style guide to code. It ensures consistent formatting across the Datadive codebase, regardless of individual developer preferences. As an opinionated formatter, Prettier limits configuration options to keep setup simple and avoid unnecessary debates and bikeshedding.

The term "bikeshedding" refers to the tendency of people to spend an inordinate amount of time discussing trivial or unimportant details which is often seen in software development when developers focus on minor details like code formatting instead of addressing critical issues like architecture or performance. [26]

The Datadive Prettier configuration is stored at the root of the Datadive repository in `prettier.config.js`. Notable deviations from the default Prettier configuration include using single quotes for strings, adding trailing commas in arrays and objects, and omitting semicolons at the end of statements. Single quotes have become the standard in the JavaScript ecosystem, trailing commas simplify code reviews by ensuring that adding or removing an item from an array or object does not affect surrounding lines. The absence allows for easier line swapping without worrying about semicolon placement. [27] [28] [21]

4.2. Tests

The Datadive codebase follows the principle of locality of behavior, which groups related code together to enhance understanding and maintenance. This principle is based on the idea that code that works together should be close together. Following this principle, tests are located in the same directory as the code they test, ensuring that tests are easy to find and maintain. The Datadive codebase uses Vitest as the testing framework for unit and integration tests. The test file is named after the file it tests, with a `.test.ts` extension. For example, tests for `parse-cookie.ts` would be in the same directory in a file named `parse-cookie.test.ts`. This naming convention makes it easy to associate tests with the code they cover and ensures that tests are located near the code they test.

4.3. Functional Programming

While Datadive does not enforce a strict functional programming paradigm, it encourages functional programming principles where appropriate. Functional programming emphasizes the use of pure functions and immutability to improve code quality and maintainability. Pure functions have no side effects and always return the same output for the same input, making them easier to reason about and test. Immutability ensures that data cannot be changed after it is created, reducing the risk of bugs caused by unintended modifications. [29]

Datadive employs functional programming principles by minimizing the use of classes and prioritizing functions and modules. Classes are used sparingly and only when necessary, such as for creating instances of objects with state or for custom errors. Many files in Datadive export only a single function and may include non-exported utilities to break the implementation into smaller parts. This approach mirrors the structure of React applications, where files often export a single component. Modules encapsulate related functionality and promote code reuse. By adhering to functional programming principles, Datadive aims to produce code that is predictable, maintainable, and testable.

4.4. Error Handling

Error handling is a crucial aspect of writing robust and reliable software. Proper error handling ensures that applications respond gracefully to unexpected conditions and provide meaningful feedback to users. Modern languages like Swift and Rust support typesafe errors, a highly requested feature in TypeScript. Typesafe errors enable developers to define a set of error types that can be thrown by a function or method, see figure 10. [30] [31] [32] [33]

```
function divide(a: number, b: number) throws DivisionByZeroError: number {
  if (b === 0) {
    throw new DivisionByZeroError("Can't divide by zero");
  }
  return a / b;
}
```

Fig. 10: Function that uses an non-existent syntax to include the thrown error in it's signature.

This approach enables developers to identify potential errors and manage them effectively. Additionally, the TypeScript compiler can enforce the handling of all possible errors. However, due to its design and compatibility with JavaScript, TypeScript does not support type-safe errors and likely never will. Instead, Datadive uses the neverthrow package to handle errors in a type-safe way. Neverthrow adopts an "errors as values" approach to error management. Instead of throwing exceptions, functions return a `Result` type that can be either `Ok` or `Err`. See the example in figure 11. [30]

```
function divide(a: number, b: number): Result<number, DivisionByZeroError> {
  if (b === 0) {
    return err(new DivisionByZeroError("Can't divide by zero"))
  }
  return ok(a / b)
}

const result = divide(10, 0)
if (result.isErr()) {
  console.error(result.error.message)
} else {
  console.log(result.value)
}
```

Fig. 11: Function that returns a `Result` type to manage errors in a type-safe manner.

This is essentially an implementation of the either monad, which is a functional programming construct that represents a value that can be either a success or a failure [33]. It is commonly used for error handling in functional programming languages. By using neverthrow, Datadive ensures that errors are managed consistently and safely throughout the codebase.

To maintain consistency in error handling across the codebase, Datadive defines custom error classes for common scenarios. These classes extend the `DatadiveError` class provided by the `@datadive/utils` package. Each error class must specify a unique error code, which should be a lowercase string.

```
class DivisionByZero extends DatadiveError<
  'division_by_zero',
  { numerator: number; denominator: 0 }
> {
  public readonly code = 'division_by_zero'
}

const error = new DivisionByZero(
  // Required error data
  { numerator: 1, denominator: 0 },
  // Optional message
  `Can't divide by zero`,
  // Optional cause
  { cause: undefined },
)
```

Fig. 12: Example custom error class that extends the `DatadiveError` *class.*

As in the example in figure 12, the errors may provide additional context and information by including a `data` property, which must be a JSON-serializable object. This `data` property often contains details such as the input that triggered the error or the context in which it occurred. Errors may also have a `message` property, used for logging purposes, which should be a concise, human-readable description of the error and its cause. The options passed to a Datadive error class include a `cause` property, indicating the error that led to the current one. This property is useful for tracking the chain of errors that resulted in the current error, especially when a third-party library's error is wrapped in a custom error.

4.5. Dependency Injection

Dependency injection is a design pattern that promotes loose coupling between components by injecting dependencies from the outside instead of creating them internally. This approach makes components more modular and easier to test, as dependencies can be mocked or replaced with stubs. Datadive employs a straightforward, functional approach to dependency injection for managing dependencies between modules and components. By convention, dependencies are passed as the first argument to functions, as illustrated in figure 13. [34] [35]

```
function divide(
  dependencies: { logger: (message: string) => void },
  a: number,
  b: number,
): Result<number, DivisionByZeroError> {
  dependencies.logger(`Dividing ${a} by ${b}`)
  if (b === 0) {
    return err(new DivisionByZeroError("Can't divide by zero"))
  }
  return ok(a / b)
}
```

Fig. 13: Function using dependency injection to manage the `logger` *dependency.*

This method allows for easy replacement or mocking of dependencies during testing. For instance, when testing the `divide` function, console output may not be desired. By passing a mock logger function as a dependency, the test can capture and verify log messages without affecting the console. See figure 14.

```
const logger = vi.fn()
const result = divide({ logger }, 10, 0)

expect(logger).toHaveBeenCalledWith('Dividing 10 by 0')
```

Fig. 14: Example test using a mock logger provided using dependency injection.

Figure 15 shows how the `console.log` function can be injected as a dependency to create a new function that can be used during runtime.

```
const runtimeDivide = (...params: Tail<Parameters<typeof divide>>) => {
  return divide({ logger: console.log }, ...params)
}
```

Fig. 15: Injecting a dependency to a function during runtime.

Since manually injecting dependencies requires more complex type helpers, which can be verbose and confusing for less experienced contributors, Datadive offers utility functions to simplify this process. See figure 16 for an example. These utility functions ensure that only `Result` or `ResultAsync` for asynchronous operations can be returned from the function. This enforces both dependency injection conventions and a consistent error handling strategy.

```
interface MathInjection {
  logger: (message: string) => void
}

const [define, inject] = createInjectionUtilities<MathInjection>()

const divide = define((dependencies, a: number, b: number) => {
  dependencies.logger(`Dividing ${a} by ${b}`)
  if (b === 0) {
    return err(new DivisionByZeroError("Can't divide by zero"))
  }
  return ok(a / b)
})

const runtimeDivide = inject({ logger: console.log }, divide)
```

Fig. 16: Example, using utility functions to simplify dependency injection.

4.6. Factory Functions

Factory functions are functions that create and return objects. They are commonly used to encapsulate object creation logic and provide a way to customize object creation without exposing the underlying implementation. [36]

Datadive uses factory functions to create instances of objects with complex initialization logic and to abstract object creation from the calling code. Like in figure 17, factory functions typically have the `create` prefix and accept configuration options as arguments. Datadive packages that rely heavily on dependency injection use factory functions to create objects that include all their methods with injected dependencies.

```
const createUser = define((dependencies, name: string) => {
  dependencies.db.insertUser({ name })
})

const createNotebook = define((dependencies, path: string) => {
  dependencies.db.insertNotebook({ path })
})

export function createCore(config: { db: { url: string } }) {
  const db = createDb(config.db.url)
  return {
    createUser: inject({ db }, createUser),
    createNotebook: inject({ db }, createNotebook),
  }
}
```

Fig. 17: Example factory function used to create an object of methods with injected dependencies.

4.7. Enum Like Objects

Enum-like objects are collections of related constants. They resemble enums in other languages but are implemented as objects. This pattern is common in TypeScript, where enums can have certain pitfalls. One issue is that their transpilation to JavaScript may lead to unexpected behavior, particularly when iterating over the values. As shown in figure 18, enum-like objects are typically constant objects with string primitive values. They also have a type of the same name that is a union of the object's values [37]. In Datadive, enum-like objects define a set of related values used throughout the codebase. For example, the `LandlordTable` and `TenantTable` objects specify the names of all tables available in the landlord and tenant databases, respectively.

```
const LandlordTable = {
  User: 'user',
  Notebook: 'notebook',
} as const

// 'user' | 'notebook'
type LandlordTable = ValueOf<typeof LandlordTable>

function printTable(table: LandlordTable) { // used as type
  console.log(table)
}

printTable(LandlordTable.User) // used as value
printTable(LandlordTable.Notebook) // used as value
```

Fig. 18: Example enum-like object defining the names of tables in the landlord database.

5. Technologies

As already mentioned, maintainability is the most important requirement for the Datadive codebase. The selection of technologies was made in such a way that it presents a low entry barrier for contributing code to Datadive. This approach involves using a limited number of programming languages and technologies. It also prioritizes tools that assist developers in the development process, such as those that enable static analysis of the code to prevent bugs before they are committed and enforce best practices.

Other important factors in technology selection include the adoption rate and popularity within the JavaScript ecosystem. These factors are significant because widely adopted technologies tend to be more stable, offer more learning resources, and are more likely to receive ongoing maintenance. Additionally, choosing technologies commonly used in the industry can provide developers contributing to Datadive with valuable experience that extends beyond the project's scope. To assess the adoption rate and popularity of technologies, this thesis considers the Stack Overflow Developer Survey 2024, the State of JavaScript 2022, and the GitHub Octoverse 2023. [38] [39] [40]

The following sections describe the major technologies used in the Datadive codebase and the reasons for their selection. Please note that not all dependencies used in Datadive are listed here, since some are only relevant for specific parts of the codebase. The technologies described here are those that have a significant impact on the overall architecture and development process of Datadive.

5.1. Typescript

Since Datadive is an application with a very interactive frontend, one of the earliest requirements was that the frontend should be a SPA. Thus, the Datadive client app, which is the part of Datadive that provides the user interface, is executed in the browser. Although modern browsers theoretically allow the execution of WebAssembly[10] (WASM) compiled code, which can be written in a variety of different languages, all modern frameworks for creating SPA's use JavaScript as the primary language. Despite WebAssembly being a promising technology, it is not yet widely adopted in the industry, and the tools and libraries available for it are not as mature as those for JavaScript. Additionally, WebAssembly is not intended to replace JavaScript in the browser but to complement it. [41] [38]

JavaScript, or more precisely ECMAScript, is a high-level, interpreted programming language. Originally developed in the mid-1990s by Netscape, JavaScript has for many years been the most used language in software development. However, as a dynamically typed language, variable types are determined at runtime rather than at compile time. While this flexibility can allow for quicker development, it can also lead to potential runtime errors since type-related issues may not be detected until the code is executed. [38] [42]

To enable static code analysis tools and reduce type-related bugs, Datadive uses TypeScript. Introduced by Microsoft in 2012, TypeScript is a statically typed superset of JavaScript that gets transpiled to JavaScript by the TypeScript Compiler (`tsc`). TypeScript automatically infers type information from both the code and type annotations. This type information is utilized by IDE integrations and `tsc` to perform static analysis of the codebase. It can also allow for quicker contributions from developers who are unfamiliar with the codebase. Although developers need to learn TypeScript's type annotation features, this understanding provides insights that can help in comprehending the codebase and ensuring that changes do not unexpectedly introduce bugs. Writing code in TypeScript and transpiling it to JavaScript is supported by all web frameworks considered for the Datadive frontend. In the recent years TypeScript has gained popularity in the industry, as it is used by many large companies and open-source projects. The State of JavaScript 2022 survey reports that 68.4% of respondents have used TypeScript. The GitHub Octoverse 2023 report lists TypeScript as the 3rd most popular language on GitHub, with over 3 million repositories using it. The Stack Overflow Developer

Survey 2024 reports that 38.5% of respondents have done extensive development work in TypeScript, making it the 4th most used language in the survey. [43] [38] [39] [40]

The decision to use TypeScript in the frontend strongly supports its adoption for Datadive's backend development as well. Since the introduction of Node.js, the first popular JavaScript server runtime, in 2009, JavaScript has gained significant popularity for backend development. Later, TypeScript also emerged as a favored choice in this area. This growth is supported by a wide range of libraries and frameworks in the server-side JavaScript ecosystem. Using TypeScript for both the frontend and backend facilitates code sharing between the two parts of the application, including data structures and utility functions. Most importantly, this approach allows developers to work on both parts of Datadive without needing to switch between different programming languages. [38] [39]

5.2. Bun

In recent years, several new runtimes[11] where introduced to the server-side JavaScript ecosystem. Some of these are designed for specific contexts, such as serverless environments or microservices, while others serve more general purposes and can be applied to a wide range of applications. One notable general-purpose runtime is Bun, which emphasizes performance and efficiency by integrating a JavaScript engine, a test runner, a script runner, a bundler, and a package manager into a single tool. Bun is designed to optimize execution speed and minimize startup times, supporting both JavaScript and TypeScript applications. It incorporates modern JavaScript features and aims to streamline development workflows by consolidating various tools. [44] [45]

The choice of using Bun as the runtime for the Datadive backend aims to create a more cohesive environment for developers by reducing the number of tools needed to set up a functional development environment or deploy Datadive.

5.3. Kubernetes

Kubernetes is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. It was originally developed by Google and is now maintained by the Cloud Native Computing Foundation (CNCF). Kubernetes provides a robust infrastructure for deploying and managing containerized applications in production environments, offering features such as load balancing, auto-scaling, and self-healing. Kubernetes is widely used in the industry for deploying microservices, web applications, and other cloud-native workloads. [38] [46]

Datadive leverages Kubernetes to manage the deployment of Jupyter servers for each user through JupyterHub. Kubernetes provides the necessary infrastructure to manage the lifecycle of Jupyter servers, including starting, stopping, monitoring, and resource allocation. This approach allows Datadive to provide a secure and isolated environment for users to work on data analysis projects.

5.4. Hono.js & Related Libraries

Hono.js is a web framework designed for building web applications and APIs, specifically optimized for serverless and edge computing environments. Built with TypeScript, it emphasizes type safety and adheres to web standards, which helps flatten the learning curve for developers who are new to the framework. Its design incorporates familiar concepts, such as the middleware pattern and route handler definitions from other popular frameworks. This and it's adherence to web standards makes it easier for developers to transition to Hono.js without having to learn an entirely new paradigm. [47]

One of the key advantages of Hono.js is its flexibility in deployment. It can be used in various environments, including traditional servers, serverless platforms, edge computing services and of course in the context of Bun applications. This versatility means that if the deployment strategy for Datadive changes, the codebase can be adapted without needing to rewrite the entire application. [47] [48]

The other reason Datadive uses Hono.js is its integration with `@asteasolutions/zod-to-openapi`. This library allows writing OpenAPI specifications in TypeScript, which can be used to provide types for Hono.js routes using `@hono/zod-openapi`, validate incoming requests using `@hono/zod-validator`, and generate an API client[12] for the frontend. This integration ensures that the API specification is always up-to-date with the codebase, reducing the risk of inconsistencies between the API and the implementation. [49] [50]

5.5. libSQL

libSQL is an open-source fork of SQLite that enhances its capabilities with modern features while maintaining full compatibility. It introduces a server component for remote database connections, an `ALTER TABLE` extension for modifying column types and constraints, and several other improvements. This combination allows Datadive to benefit from SQLite's advantages, such as using an individual database per tenant or creating an in-memory database for testing, while also leveraging distributed databases for production. [51]

Additionally, libSQL supports local replicas, which may enable offline features in Datadive in the future. This would allow users to work on their projects without an internet connection and synchronize their changes once they are back online. [52]

5.6. Kysely & Kysely Codegen

Kysely is a SQL query builder for TypeScript that provides a type-safe and composable API for constructing SQL queries. It supports various SQL dialects, including SQLite, PostgreSQL, MySQL and libSQL. Queries are type checked using TypeScript's type system, ensuring that queries are syntactically correct and type-safe at compile time. These type checks are based on types provided by the developer which describe the database structure. This approach helps prevent runtime errors and makes it easier to refactor queries as the codebase evolves. [53]

Kysely fits well in the objective of datadive to use strict type checks and static analysis to prevent bugs before they are committed. It's API is also designed to be similar to SQL syntax, which can make it easier for developers familiar with SQL to write queries in TypeScript.

Kysely Codegen generates the types provided to Kysely from a database schema. It works by connecting to a database and using Database introspection to extract the schema information, which is then used to generate TypeScript types for tables and columns. This process ensures that the types used in Kysely queries are always in sync with the database schema, reducing the risk of type mismatches and inconsistencies between the codebase and the database. [54]

5.7. React

React (see react.dev) is a JavaScript library for building user interfaces, developed by Meta (formerly Facebook) in 2013. It employs a declarative programming model, allowing developers to describe the desired state of the UI while React updates the DOM to match that state. React is known for its component-based architecture, which enables the creation of reusable UI components. [55]

React's popularity and extensive ecosystem of libraries and tools make it a suitable choice for building the Datadive frontend. Its component-based architecture aligns well with the modular design of the frontend, allowing developers to create reusable components for different parts of the user interface. Additionally, React's strong community support and active development make it a reliable choice for long-term maintenance and scalability. [38] [39]

5.8. TanStack Router and TanStack Query

TanStack Router and TanStack Query are libraries developed by Tanner Linsley that provide routing and data fetching capabilities for React applications.

TanStack Router is a flexible and extensible routing library that allows developers to define routes and handle navigation in React applications. It supports nested routes, route parameters, and route guards, enabling developers to create complex routing structures with ease. [56]

TanStack Query is a data fetching library that simplifies fetching, caching, and updating data in React applications. It provides a declarative API for fetching data from APIs and managing the data lifecycle, including caching, pagination, and optimistic updates. TanStack Query integrates seamlessly with TanStack Router, allowing developers to fetch data based on the current route and update the UI in response to data changes. [57]

5.9. Vite & Vitest

Vite is a modern build tool for front-end development that focuses on speed and simplicity. It leverages native ES modules to provide fast build times and instant hot module replacement (HMR) during development. Vite supports various front-end frameworks, including React, Vue, and Svelte. It offers a streamlined development experience with features such as pre-configured development servers, optimized production builds, and built-in support for TypeScript and CSS preprocessors. [39] [58]

Vitest is a unit testing framework specifically designed to integrate with Vite. It provides an efficient and high-performance testing experience by utilizing Vite's architecture and optimizations. Vitest leverages Vite's on-demand compilation and native ES module support, allowing tests to run without a separate bundling process. [39] [59]

Given the choice of Vite as frontend build tool, Vitest is the natural choice for testing in the datadive frontend. To keep the amount of technologies used in the Datadive codebase low, Vitest is also used for testing the backend.

5.10. Tailwind CSS

Tailwind CSS is a utility-first CSS framework that provides a set of pre-built utility classes for styling web applications. It allows developers to create custom designs without writing custom CSS by combining utility classes to style elements. Tailwind CSS is designed to be customizable, enabling developers to create unique designs by configuring the framework's utility classes. Since it's highly flexible and efficient, Tailwind CSS has been widely adopted by developers and companies worldwide, including major tech organizations like GitHub, Netflix, and Shopify, making it one of the most popular CSS frameworks in modern web development. [60] [61]

The decision to use Tailwind CSS for the Datadive frontend aims to streamline the styling process and provide a consistent design system across the application. Tailwind CSS's utility classes restrict the number of custom styles available to developers, reducing the risk of inconsistent styling and making it easier to maintain and update the design system. Additionally, writing CSS using utility classes can lighten the cognitive overhead of styling elements by abstracting away the need to write custom CSS selectors and consider about class naming conventions.

5.11. Astro.js

Astro.js is a modern web framework designed to simplify the creation of high-performance, content-focused websites. It employs a novel architecture that combines the best features of server-side rendering (SSR) and static site generation (SSG). Astro.js introduces "partial hydration," allowing developers to selectively hydrate interactive components on the client side while rendering most of the page content on the server. This approach reduces the amount of JavaScript sent to the client, leading to faster page loads and improved performance. [62]

Datadive utilizes Astro.js, specifically the documentation template Starlight, for its documentation. This choice somewhat deviates from the goal of using a minimal number of technologies, as Astro.js is another web framework. However, as a static site generator, Astro.js is a better fit for documentation than React, which is the frontend framework used for the Datadive client app. Most Datadive maintainers will only need to write Markdown files. Astro.js then uses these files as input to generate the documentation, meaning most developers working on Datadive won't need to learn Astro.js to contribute to the documentation. [63]

5.12. Zod

Zod is a TypeScript-first schema declaration and validation library that provides a simple and expressive API for defining data schemas. It allows developers to define data structures using TypeScript, infer the types of the validated data to use them for type checking and validate data against those schemas at runtime. [64]

Datadive relies heavily on Zod for defining data structures. In the backend, zod schemas are used to validate incoming requests and responses, ensuring that the data conforms to the structure defined in the Datadive API Specification. This validation helps prevent malformed data from entering the system and ensures that the data is consistent and predictable. In the frontend, zod schemas are used to validate user input and responses from the backend. By using Zod, Datadive enforces data integrity and consistency across the application, reducing the risk of runtime errors and data corruption.

5.13. neverthrow

Neverthrow is a TypeScript library that provides a approach to error handling that is inspired by functional programming and error handling in the Rust programming language. It introduces the `Result` type, which represent the result of an operation as either a success value or an error value. By using the `Result` type, developers can handle errors in a composable and type-safe manner. [65]

Datadive uses neverthrow to handle errors in the backend. Returning `Result` types from core methods and repositories, errors are part of a methods type signature, making it easier to reason about which errors can happen and encouraging explicit error handling.

5.14. Radix UI

Radix UI is a collection of low-level, accessible UI components for React. It offers a set of unstyled components that developers can use to create custom designs while ensuring accessibility and usability. [66]

Datadive utilizes Radix UI to develop accessible and customizable UI components for the client app. The unstyled nature of Radix UI components enables Datadive contributors to create custom designs. This choice was made because Datadive will feature a complex, highly interactive frontend with custom UI components not available in styled component libraries. The initial effort required to style Radix UI components can be offset by using templates like shadcn/ui, which provide pre-styled components based on Radix UI and other unstyled component libraries. [67]

5.15. ESLint

ESLint is a static code analysis tool for identifying problematic patterns in JavaScript code. It enforces coding standards and best practices, helping developers write clean, consistent, and maintainable code. ESLint is highly configurable, allowing developers to customize the rules and plugins used to analyze the codebase. [68]

Datadive uses ESLint to enforce coding standards and best practices in the codebase. By configuring ESLint with a set of rules that align with Datadive's coding standards, developers can identify and fix potential issues before they are committed. ESLint also integrates with IDEs and text editors, providing real-time feedback on code quality and helping Datadive contributors to adhere to the established coding standards.

5.16. Prettier

Prettier is an opinionated code formatter that automatically formats code to ensure consistency and readability. It enforces a consistent code style by parsing the code and reformatting it according to predefined rules. Prettier supports various programming languages, including JavaScript, TypeScript, CSS, and HTML. [69]

Datadive uses Prettier to maintain a consistent code style across the codebase. By configuring Prettier with a set of formatting rules, developers can ensure that the code is formatted consistently and adheres to the established coding standards.

Prettier integrates with IDEs and text editors, providing automatic code formatting and reducing the time spent on manual formatting tasks.

5.17. Turborepo

Turborepo is a build system specifically designed for managing JavaScript and TypeScript monorepos. It addresses the challenges associated with such repositories by implementing efficient build and caching mechanisms. Turborepo uses a task graph to identify the minimal set of tasks required for a build, thus optimizing the build process and reducing redundant work. This task graph enables intelligent task scheduling, which can significantly decrease build times in large-scale projects. By caching outputs, Turborepo avoids unnecessary recomputation, enhancing build efficiency. This system is particularly advantageous for development teams working within monorepos, as it supports a more efficient workflow and can enhance the overall developer experience. [70]

Datadive uses Turborepo to manage the monorepo structure of the codebase. By leveraging Turborepo's build system, Datadive can optimize the build process, reduce build times, and enhance the development workflow. This choice aligns with Datadive's focus on maintainability and efficiency, as Turborepo provides a scalable solution for managing a large codebase.

6. Third Party Services

Datadive uses third party services to avoid implementing complex functionality that is not core to the platform. This chapter provides an overview of the services used by Datadive, explaining their purpose, structure, and interactions.

6.1. Turso

Turso is a distributed database system that builds upon the core functionality of libSQL which is an open source fork of SQLite. It takes the simplicity and ease-of-use of the SQLite database engine and extends it to a distributed architecture. Datadive uses libSQL Databases hosted on Turso to store and manage data for the platform. The main reason is that Turso provides a full featured, scalable database system that can handle large amounts of data and high query loads. It offer features like point-in-time restore, multi-database schemas and fault tolerance which are essential for a platform like Datadive. These features are complicated to implement and maintain, so using Turso allows the Datadive team to focus on building the core functionality of the platform. [52]

6.2. Resend

Resend is an email service designed specifically for developers to build, test, and send transactional emails at scale. The platform aims to provide a reliable and scalable email delivery solution, focusing on deliverability and compliance to ensure successful email communication for its customers.

Email delivery is a critical part of the Datadive platform, as it is used to send notifications, alerts, and reports to users. Resend provides a simple API that allows Datadive to send emails programmatically, without having to manage email servers or infrastructure. The platform also offers features like tracking, analytics, and reporting, which help the Datadive team monitor the performance of their email campaigns and improve deliverability. [71]

7. Packages

This chapter provides an overview of the packages of the Datadive platform, explaining their purpose and structure.

Except for some configuration packages, all packages have a main entrypoint that exports the main functionality of the package. Additionally, some packages have another entrypoint that exports the error classes contained in the package. This is done to prevent type errors when exporting functions that may return the error classes from other packages.

7.1. The `@datadive/db` Package

The `@datadive/db` package contains database-specific code, including migrations, seeds, and functions to initialize the database connection. It is separated into tenant and landlord database code. Additionally, it includes a CLI for common development tasks, such as running migrations or seeding the database.

- `execute` - Function that executes a query on the database and returns a result either containing the data or an error.
- `executePaginated` - Function that executes a query on the database and returns a result containing the paginated data or an error.
- `executeTakeFirst` - Function that executes a query on the database and returns a result containing the first row of the data or an error if the query failed or no data was found.
- `createDatabaseClient` - Function that creates a new database client instance. The database client is used to communicate with a database.
- `createLandlordKysely` - Factory function that creates a new Kysely instance using a database client. Use the kysely instance to build queries which may be executed using the execution functions.
- `LandlordTable` - Enum like object that contains the names of all tables that are available in the landlord database. Use this object to reference tables in queries.

- `createTenantKysely` - Factory function that creates a new Kysely instance using a database client. Use the kysely instance to build queries which may be executed using the execution functions.
- `TenantTable` - Enum like object that contains the names of all tables that are available in the tenant database. Use this object to reference tables in queries.
- `DbError` - Module that contains all error classes that may be returned when using the database.
- `LandlordDatabaseSchema` - Type that represents the schema of the landlord database.
- `TenantDatabaseSchema` - Type that represents the schema of the tenant database.
- `Selectable` - Utility type to get the type of a data that can be selected from a table in either the landlord or tenant database. This can be used to get the the type of e.g. the result of a select query on the landlord user table by using `Selectable<LandlordTable, typeof LandlordTable.User>`.
- `Insertable` - Utility type to get the type of a data that can be inserted into a table in either the landlord or tenant database. This can be used to get the the type of e.g. the data that can be inserted into the landlord user table by using `Insertable<LandlordTable, typeof LandlordTable.User>`. The insertable type may not be the same as the selectable type, as some columns may be automatically generated by the database.
- `Updatable` - Utility type to get the type of a data that can be updated in a table in either the landlord or tenant database. This can be used to get the the type of e.g. the data that can be updated in the landlord user table by using `Updatable<LandlordTable, typeof LandlordTable.User>`. The updatable type may not be the same as the selectable or insertable type, as some columns may be automatically generated by the database.
- `migrate` - Function that runs database migrations.
- `getMigrationInfo` - Function that returns information about the current state of the database migrations, e.g. wether a migration has been run or not.
- `createDatabase` - Function that creates a new database. This function is used to

create the tenant databases when a new tenant is created.

7.2. The `@datadive/email` Package

The `@datadive/email` package contains code for sending emails. It does not contain any templates, as these are stored in the `@datadive/core` package co-located with the code that uses them. This package is basically a thin abstraction around Resend which is the service used to send emails. It's main entrypoint has the following exports:

- `EmailError` - Module that contains all error classes that may be returned when using the email functions.
- `createEmail` - Factory function that creates a new email object which contains all functions necessary to send emails.

7.3. The `@datadive/utils` Package

The `@datadive/utils` package includes utilities used throughout the Datadive platform. These functions are not tied to any specific part of the platform and can be utilized across multiple packages. The package offers three entry points: `@datadive/utils/browser` for browser-specific utilities, `@datadive/utils/common` for utilities applicable in both browser and server environments, and `@datadive/utils/type` for utility types. Since the package includes numerous exports, primarily simple functions or types with attached JSDoc comments, they will not be listed here.

7.4. The `@datadive/auth` Package

The `@datadive/auth` package contains code for authenticating users and managing user sessions. It provides functions handling user authentication and authorization. The package is divided into two main parts, one part for tenant authorization and one part for landlord authorization. The session handling is implemented according to the instructions of Lucia Auth. It's main entrypoint has the following exports:

- `createLandlordAuth` - Factory function that creates a new auth object which contains all functions necessary to authenticate users.
- `createTenantAuth` - Factory function that creates a new auth object which contains all functions necessary to authenticate users.
- `AuthError` - Module that contains all error classes that may be returned when using the auth functions.

7.5. The `@datadive/spec` Package

The `@datadive/spec` package contains the specifications for the Datadive HTTP API. It uses the `@asteasolutions/zod-to-openapi` to define the routes and models of the API in Typescript. The spec is used to provide validation and type safety to the backend, to generate an API client for the frontend, and to generate the OpenAPI documentation. The package exports:

- `LandlordEndpoints` - Object that contains the specification of all landlord endpoints of the API.
- `TenantEndpoints` - Object that contains the specification of all tenant endpoints of the API.
- `ApiError` - Object that contains the specifications of all errors that may be returned by the API.
- `ApiErrorName` - Enum-like object that contains the names of all errors that may be returned by the API.
- `ApiErrorCode` - Enum-like object that contains the codes of all errors that may be returned by the API.

7.6. The `@datadive/core` Package

The `@datadive/core` package contains the core functionality of the Datadive platform. It includes the main application logic, such as handling user interactions, managing data, and orchestrating the communication between the backend and the

Jupyter components. The package is divided into tenant and landlord code. It's main entrypoint has the following exports:

- `createLandlordCore` - Factory function that creates a new core object which contains all functions that implement the core functionality of the landlord part of the platform.
- `createTenantCore` - Factory function that creates a new core object which contains all functions that implement the core functionality of the tenant part of the platform.
- `CoreError` - Module that contains all error classes that may be returned when using the core functions.

7.7. The `@datadive/jupyter` Package

The `@datadive/jupyter` package includes code for communicating with the Jupyter components of the Datadive platform. It is divided into functions for interacting with the JupyterHub API and the Jupyter Server API. The package utilizes the OpenAPI specification of the Jupyter APIs to generate a client, which is then wrapped to offer a more streamlined API and semantic error classes. It's main entrypoint has the following exports:

- `createJupyterHubClient` - Factory function that creates a new JupyterHub communication client object which contains all functions necessary to interact with the JupyterHub component.
- `createJupyterServerClient` - Factory function that creates a new Jupyter Server communication client object which contains all functions necessary to interact with the Jupyter Server component.
- `JupyterError` - Module that contains all error classes that may be returned when using the functions of either client.

7.8. The Configuration Packages

The Datadive repository includes several configuration packages that are used to distribute shared configuration files across the repository. The configuration packages include `@datadive/eslint` which contains the ESLint configuration, `@datadive/storybook` which contains the Storybook configuration, and `@datadive/tsconfig` which contains the TypeScript configuration for the Datadive repository. These packages are used to ensure consistent linting, formatting, and build configurations across the Datadive codebase.

8. Applications

The Datadive repository contains three separate applications: the frontend, the backend, and the documentation. Each application is stored in a separate package in the `/apps` directory of the repository. This chapter provides an overview of each application, explaining its purpose and structure.

All applications are set up to be built using the `bun run build` command. This command compiles the TypeScript code into JavaScript and outputs it to the `dist` directory. You can then start the server with the `bun run start` command, which runs the compiled code in production mode using the configuration from the `.env.production.local` file. To start the development server with hot reloading[13], use the `bun run dev` command. This command launches the server in development mode using the configuration from the `.env.development.local` file.

8.1. The `@datadive/api` App

The API application is the backend of the Datadive platform. It provides a HTTP API for interacting with the platform that is implemented using Hono.js. The API application is responsible for handling requests from the frontend, executing data analysis tasks, and managing user data. It interacts with the tenant and landlord databases to store and retrieve data, as well as with the Jupyter components to execute data analysis workflows. The API application is designed to be scalable and extensible, allowing new endpoints and functionality to be added easily. All endpoints follow the specification defined in the `@datadive/spec` package.

```
api/
├── README.md
├── package.json
├── src/
│   ├── api-env.ts
│   ├── index.ts
│   ├── landlord/
│   │   ├── middleware/       # Landlord specific middleware
│   │   ├── routes/           # Landlord specific routes
│   │   └── shared/           # Shared landlord code
│   ├── reset.d.ts
│   ├── shared/
│   │   ├── exceptions/       # Shared exceptions
│   │   └── utilities/        # Shared utilities
│   └── tenant/
│       ├── middleware/       # Tenant specific middleware
│       ├── routes/           # Tenant specific routes
│       └── shared/           # Shared tenant code
├── tsconfig.build.json
└── tsconfig.json
```

Fig. 19: Structure of the `@datadive/api` application

As shown in figure 19, the code is divided between tenant and landlord functionality, with each part containing its own set of routes and middleware. The entrypoint of the server is located in the `src/index.ts` file, which initializes the server and sets up the routes and middleware of both the tenant and landlord parts. It also serves the build output of the React web application, which is the fallback response for all requests that do not match an API endpoint.

The configuration for the development mode of the API application is stored in the `.env.development.local` environment file. This file contains the configuration for the landlord database connection, the port on which the server should run, and other configuration that is specific to the development environment. The environment files are loaded and validated before the server is started, ensuring that the application runs with the correct configuration. The schema for the environment files is defined in the `src/api-env.ts` file, which uses the T3 Env (see env.t3.gg) package and Zod (see zod.dev) for validation. An example of the environment file is provided in the `.env.example` file.

8.2. The `@datadive/web` App

The web application is the frontend of the Datadive platform. It's implemented using React, Vite and TanStack Router. The web application is an SPA that includes both the landlord and tenant interfaces. The landlord interface is used by administrators to manage users, projects, and notebooks, while the tenant interface is used by users for data analysis tasks. The web application interacts with the Datadive HTTP API to handle user interactions. It uses TanStack Query and an API client generated from the OpenAPI specification to communicate for data fetching.

The datadive web application is very minimal, only a demo application to show the concept of the platform and provide a skeleton for future development. It is not intended to be a full-featured application, but rather a starting point for building a more complex frontend with rich functionality.

The code is divided between tenant and landlord functionality, with each part containing its own set of pages and components. The entrypoint of the application is located in the `src/main.tsx` file. Pages are stored in the `src/routes` directory, according to the conventions of the file based routing system of TanStack Router. Landlord routes are located in the `src/routes/landlord` directory, while tenant routes are located in the `src/routes` directory. Pages are tightly coupled to TanStack router, using it's features like `useRoute` and `useParams` to access the current route and parameters. Components are stored in the `src/components` directory, with shared components located in the `src/components/shared` directory, landlord specific components in the `src/components/landlord` directory, and tenant specific components in the `src/components/tenant` directory. Components are designed to be reusable and composable, following the principles of component-driven development. They should not use any TanStack Router specific features but instead rely on props and context to access data and functionality.

The application can be build using the `bun run build` command, which creates a production build of the web application and outputs it to the `dist` directory. The production build can be started using the `bun run start` command, which serves the compiled code using the configuration from the `.env.production.local` file. The production build is also served by the API application.

8.3. The `@datadive/docs` App

The documentation application is a static site that provides information about the Datadive platform. It is implemented using Astro.js and more specifically Starlight template (see starlight.astro.build). The documentation application is stored in the `/apps/docs` directory of the repository.

The folder structure of the documentation application is determined by the folder structure of Starlight. The documentation is stored

9. Setup Development Environment

The setup of the development environment contains several steps that need to be followed to ensure that the Datadive platform can be developed and tested locally. This chapter provides a step-by-step guide on how to set up the development environment for the Datadive platform. The guide covers the installation of the required tools, the configuration of the development environment, and the setup of the project repository.

9.1. Jupyter Hub

The Datadive platform uses JupyterHub as a component of the backend. It is used to manage Jupyter Servers for users, which in turn are used to manage and execute notebooks. Subsequently, JupyterHub needs to be installed and configured to run the Datadive platform. Since JupyterHub is run in Kubernetes, the setup of a local Kubernetes cluster is required. To set up a local Jupyter Hub environment, follow the steps in the Zero to JupyterHub guide for a local Kubernetes Cluster in Minikube. [72]

9.2. VSCode

VSCode is the recommended code editor for developing the Datadive platform. It provides a rich set of features for code editing, debugging, and version control. It is possible to use other code editors, but the Datadive repository contains settings and extensions for VSCode that make development easier. To install Visual Studio Code, download the installer from the official website and follow the installation instructions for your operating system. [73]

9.3. Bun

Bun is the runtime environment for the Datadive platform. It is also the package manager used to manage the dependencies of the platform. To install Bun, follow the installation instructions on the official website. [45]

9.4. Turso Account

The Datadive platform utilizes Turso as a third-party service for distributed databases. To use Turso, create a free account on the Turso website (see turso.tech). Datadive requires one database for the landlord and one for each tenant. The free account has some limitations: as of this writing, it supports 500 databases with a maximum of 9GB of storage, 1 billion rows read, and 25 million rows written per month. However, these limits are more than sufficient for the development and testing purposes of the Datadive platform.

9.5. Resend Account

Resend is a third-party service used by the Datadive platform to send transactional emails. To use Resend, create an account on the Resend website. A free account allows sending up to 3000 emails per month and 100 emails per day. This limit is sufficient for the development and testing purposes of the Datadive platform. [71]

9.6. Repository Setup

The Datadive repository is hosted on GitHub and contains all the code for the platform. To clone the repository, run the following command:

```
git clone https://github.com/lucaschultz/datadive.git
```

This command will create a local copy of the repository in a directory named `datadive`. Navigate to this directory to start working on the Datadive platform. The Datadive platform has several dependencies that need to be installed before you can start developing. To install the dependencies, run the following command in the root directory of the repository:

```
bun install
```

This command will install all the required packages and tools needed to run the Datadive platform. Once the installation is complete, build all packages by running:

```
turbo build
```

This might take a while as it compiles all the TypeScript code into JavaScript. It is necessary because some of the packages depend on the build output of other packages.

9.6.1. Environment Files

After the build is complete, set up the `.env` files for development. Datadive needs two env files for development, one for the API and one for the web app. The repository contains example files that you can copy and modify. At the root of the repository, run:

```
cp apps/web/.env.development.example apps/web/.env.development.local
cp apps/api/.env.development.example apps/api/.env.development.local
```

You don't need to change anything in the frontend env file. The backend file needs to be modified. Open the `apps/web/.env.development.local` file in your code editor. Replace `<TURSO_API_KEY>` with a valid API key for the Turso Platform API. This API key can be created either in the Turso web interface or using the turso CLI by running:

```
turso auth api-tokens mint <api-token-name>
```

Replace `<api-token-name>` with a name for the API token. The command will return an API key that you can use in the `.env.development.local` file. Next, replace `<RESEND_API_KEY>` with a valid API key for the Resend API. This API key can be created in the Resend web interface.

The Datadive repository includes a CLI tool that can be used for common development tasks such as creating a landlord or tenant, running migrations, or generating a new app key. An app key is used to encrypt sensitive data in the database and is required for the development environment. To generate a new app key, run:

```
bun run cli make:app-key --env=apps/web/.env.development.local --force
```

This command will generate a new app key and store it in the `apps/web/.env.development.local` file. The `--force` flag is used to overwrite the existing `<APP_KEY>` placeholder or app key. The app key is used to encrypt sensitive data in the database and should be kept secret. If you change the app key, you'll need to reset the landlord and delete all tenants. Therefore, it is recommended to save a backup of the app key in a secure location like a password manager.

The Datadive platform requires a landlord to manage tenants. To create a landlord, run the following command in the root directory of the repository:

```
bun run cli make:landlord --env=apps/web/.env.development.local development
```

This command will create a landlord database using the Turso API, run the landlord migrations, and print the connection details to the console. The connection details include the database URL and token, which are required to connect to the landlord database. Replace the `<LANDLORD_DATABASE_URL>` and `<LANDLORD_DATABASE_TOKEN>` placeholders in the `apps/web/.env.development.local` file with the connection details. The last step is to create a user for the landlord. To create a user, run the following command in the root directory of the repository:

```
bun run cli seed:landlord --env=apps/web/.env.development.local
```

This command will create a user with the email `developer@datadive.app` and the password `password`. You can use these credentials to log in to the landlord interface of the Datadive platform.

9.7. Running Datadive

To start the Datadive platform, run the following command in the root directory of the repository:

```
bun run dev
```

This command will start the Datadive platform in development mode. The platform consists of the API and the web application. The API will be available at `http://localhost:3000` and the web application at `http://localhost:3001`. You can access the web application in your browser to start developing and testing the Datadive platform.

9.8. Creating a Tenant

To create a tenant using the CLI, run the following command in the root directory of the repository:

```
bun run cli make:tenant --env=apps/web/.env.development.local dev
```

This will create a tenant database using the Turso API and run the tenant migrations. To access the tenant interface, open localhost:3001/landlord/dev. You can log in with the email `developer@datadive.app` and the password `password`.

10. Outlook

This thesis presents the architecture, code structure, and development environment of the Datadive platform. Although the current state is far from a complete data analysis platform, it lays a foundation for future development. This chapter reviews the platform's current status and outlines the next steps needed to make it fully functional for researchers.

10.1. Challenges and Limitations

Exploring and implementing a viable architectural framework for this thesis project proved complex and time-consuming. While the initial plan aimed for a comprehensive demonstration, time constraints ultimately limited the scope of functionality. The architecture design underwent several iterations, and the exploratory prototypes built to assess the architecture's feasibility consumed more time than expected. Initially, the idea was to implement an event-driven architecture that would execute code in plugins communicating with the main application through a message broker. However, this approach was abandoned in favor of using Jupyter components for code execution. Although the event-driven architecture would have provided more flexibility, the Jupyter components offered a more straightforward solution. This reduced the platform's complexity by offloading code execution to a well-established open-source project. Additionally, users can utilize the Jupyter Lab interface of the Jupyter servers, allowing them to revert to writing code in a Jupyter notebook if the Datadive GUI does not meet their needs.

The first prototype of a Jupyter-based architecture used a single Jupyter server running in a Docker container for code execution. While this approach worked well in a prototype, it did not allow for isolating user code execution in a production environment. The final architecture employs individual Jupyter servers managed by JupyterHub running in Kubernetes to execute code in isolated environments, providing a secure and scalable solution for code execution.

This architecture lays a solid foundation for the Datadive platform, offering a clear structure for future development. The repository containing the codebase is organized to facilitate ongoing development, featuring clear documentation, automatic code analysis, and a prototype that verifies the proposed solutions.

The prototype enables users to create projects and notebooks, as well as execute code using Jupyter components through the HTTP API and a limited frontend.

10.2. Next Steps

The immediate next steps for the Datadive platform involve enhancing the functionality of both the frontend and backend applications. The data model created by the migrations of the `@datadive/db` package includes all the tables necessary for the platform's core functionality, such as user authentication, project management, and notebook execution. The API definition in the `@datadive/spec` package will be extended to include all necessary endpoints for this core functionality, and these endpoints will be implemented in the `@datadive/api` package.

Since the entire data model revolves around users "owning" projects that contain notebooks, it is essential to prioritize the implementation of endpoints for user management and authentication, as these endpoints form the foundation for all other functionalities. Once these endpoints are established, the next step will be to implement the endpoints for project management, which will allow users to create, update, and delete projects.

Subsequently, the endpoints for notebook management will be refactored to enable users to create, update, and delete notebooks, as well as execute code within them. To manage notebook content, cell template endpoints will be implemented, allowing users to create, update, and delete cell templates. The final step to complete the core interactions of Datadive will be to implement the endpoints for managing notebook content, enabling users to utilize cell templates and required input data to create, update, delete, and reorder cells in notebooks.

Another crucial step to get the Datadive platform production ready is implementing a custom authenticator for JupyterHub and figuring out an deployment strategy. The authenticator should regulate access to the Jupyter servers by authenticating users against the Datadive API, allowing only authenticated users to access the Jupyter servers [74]. This is crucial for the security of the platform, as it ensures that only authorized users can execute code in the platform and that user data is protected. The deployment strategy should include any the necessary configuration for a production Kubernetes cluster, such as setting up the JupyterHub Helm chart, configuring the JupyterHub authenticator and several other settings that are necessary for a production deployment of Jupyter Hub. It is also likely, that Datadive will need a custom Jupyter Server Docker image that includes configurations to allow installing additional packages and libraries or restrict access to certain parts of the Jupyter Lab interface to ensure that users can only use the features that won't interfere with the Datadive platform [13].

The next steps for the frontend are less clear because both the design and, more importantly, user interactions require further exploration and defined requirements. The platform's core interactions involve significant complexity. Designing a user-friendly "workspace" that allows users to manage notebook cells, execute code, view results, and switch seamlessly to an IDE-like Jupyter Lab interface is challenging. Implementing dynamically generated forms from server data for cell input is complex, especially if these forms need to include UX-enhancing features like client-side data validation or drafts to save progress. The next steps for the frontend should focus on exploring various designs and user interactions, defining clear requirements, and implementing the necessary components and pages to support the platform's core interactions.

10.3. Beyond the Core Functionality

Once the core functionality of the Datadive platform is implemented, the next steps involve extending the platform with additional features. These features should be based on the requirements identified in an upcoming conceptual phase of the

Datadive project. Defining these features is beyond the scope of this thesis. However, some potential features that are likely to be needed or have been discussed during the conceptual phase of this thesis are listed in the following paragraphs.

One necessary feature is extending the platform with additional data import and export functionality. The current data model allows files to be stored on the Jupyter Server of their respective project, with references stored in the database. This approach is straightforward and effective if the files are used only within the project's notebooks. However, the Datadive platform will likely include features that require access to data outside the notebook interface. For example, users may want to preview or edit data before adding it to a project, share data between projects or share data analysis results using public links. If the data is stored on a Jupyter Server, that server must always be running for data to be accessible. Running servers is resource-intensive, and keeping all servers operational at all times is likely not scalable. Additionally, starting and stopping servers requires resources and time, which means that accessing data by frequently starting and stopping servers could lead to slow response times and is probably not scalable either. Storing data outside of Jupyter servers may present additional challenges, such as accessing files within the Jupyter Server for code execution. One solution could involve including a custom `ContentsManager` class in the Datadive Jupyter Server Docker image. With a custom `ContentsManager`, files could be stored in a shared location, such as an S3[14] bucket, and accessed by the Jupyter Server through the Contents Manager and by the Datadive backend via the S3 API. [75]

Another feature under discussion is an "assistant" that helps users select the appropriate statistical test for data analysis based on the data's shape and meta-information extracted from related sources, such as Qualtrics. This feature could be developed by expanding the concept of cell templates beyond just a collection of inputs and a code snippet with placeholders. Datadive could differentiate between complex and simple cells, with complex cells linked to hardcoded functionality that accesses project data and provides feedback to users. If this functionality were implemented generically, it could pave the way for additional features that assist users in completing data analysis tasks.

Other features that are needed are collaboration features, such as sharing projects, notebooks, and files not just with other users but also with external collaborators. Version control is another important feature that is likely to be needed. The current data model does not include versioning, which is crucial for reproducibility. Implementing version control will require changes to the data model and the backend API, as well as the frontend to support viewing and restoring previous versions of projects and notebooks.

10.4. Conclusion

The Datadive platform, though still in its early stages, establishes a foundational framework for developing a comprehensive data analysis tool. The project has made key architectural choices, such as leveraging Jupyter components and implementing a scalable infrastructure with JupyterHub and Kubernetes, which provide a solid basis for further development. While not all planned features could be implemented due to time constraints, the platform's modular design and organized codebase support ongoing enhancements. The outlined steps for implementing core functionalities, including user authentication, project management, and notebook execution, are crucial for advancing the platform. Future work will focus on addressing data management challenges and exploring additional features to assist users in data analysis. Despite current limitations, this thesis lays the groundwork for future development efforts aimed at creating a functional research tool.

Notes

1. Runtime refers to the period when a program is executed and its instructions are interpreted and processed by the computer's hardware, utilizing system resources like memory and CPU. In contrast, compile time is the stage when a high-level programming language is translated into a lower-level form, such as machine code or bytecode, that can be directly executed by the computer. This translation process, performed by a compiler, involves tasks like syntax checking, type checking, and code optimization, and occurs before the program is run.
2. An IDE (Integrated Development Environment) is a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of at least a source code editor, build automation tools, and a debugger. Some IDEs contain additional features, such as version control, code review, and profiling tools. Typically, an IDE is dedicated to a specific programming language, such as Python, Java, or C++.
3. The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems. HTTP is the foundation of data communication on the internet, where hypertext documents include hyperlinks to other resources that the user can easily access. In the context of APIs, HTTP is often used to transfer JSON or XML data between clients and servers.
4. An application programming interface (API) is a set of rules and protocols that allows different software applications to communicate with each other. APIs define the methods and data formats that applications can use to request and exchange data. In the context of web development, APIs are often used to enable communication between the frontend and backend components of a web application.
5. Docker is a platform for developing, shipping, and running applications in containers. Containers allow developers to package an application with all of its dependencies, including libraries and other binaries, and ship it as a single package. This approach ensures that the application will run consistently across different environments, regardless of the underlying system configuration. Docker containers

are lightweight, portable, and isolated, making them an popular solution for deploying applications in a variety of environments, from local development machines to production servers.

6. CRUD stands for create, read, update, and delete. It refers to the four basic operations that can be performed on data. These operations are commonly used in database management systems and web applications to manage data.
7. A single-page application (SPA) is a web application that interacts with the user by dynamically rewriting the current page rather than loading entire new pages from the server. This approach provides a more fluid user experience by avoiding the need to reload the entire page when the user interacts with the application. SPAs are typically built using JavaScript frameworks like React, Angular, or Vue.js.
8. Mocking is the process of simulating the behavior of a component or system to test its interactions with other components or systems. In the context of API development, mocking involves creating a simulated version of the API that can be used to test the frontend application without requiring a fully functional backend.
9. Kebab case is a naming convention where words are written in all lowercase letters and separated by hyphens (`-`) instead of spaces or underscores. Kebab case is commonly used in file names, URLs, and CSS classes to improve readability and consistency. For example, a kebab case file name might be `my-file-name.ts`.
10. WebAssembly (WASM) is a binary instruction format for a stack-based virtual machine. It is designed as a portable compilation target for high-level languages like C/C++/Rust, enabling code to run in the browser at near-native speed. WebAssembly is supported by all major browsers and can be executed alongside JavaScript in the same environment. [41]
11. A runtime is an environment that executes code written in a specific programming language. Runtimes provide the necessary infrastructure to run code, including libraries, APIs, and tools for compiling, interpreting, or executing code. In the context of server-side JavaScript, a runtime is a platform that allows developers to run JavaScript code on the server, enabling the development of web applications using JavaScript for both the frontend and backend.
12. An API client is a software application that interacts with an API to send and receive data. API clients are used to access the functionality provided by an API, such as retrieving data, updating information, or performing specific actions.

13. Hot reloading is a feature that automatically reloads the application when changes are made to the code. This allows developers to see the changes immediately without having to manually refresh the page. Hot reloading is a common feature in modern frontend development tools and frameworks.
14. Amazon S3 (Simple Storage Service) is a scalable object storage service offered by Amazon Web Services (AWS) designed for storing and retrieving any amount of data from anywhere on the web. It provides features such as durability, availability, and security for data storage, making it suitable for various applications including data backup, archiving, content distribution, and big data analytics. S3 organizes data into buckets, with each object identified by a unique key, enabling efficient data management and access.

Works Cited

1. Valero-Mora, P.M., Ledesma, R.D.: Graphical User Interfaces for R. Journal of Statistical Software. 49, 1, 1–8 (2012). https://doi.org/10.18637/JSS.V049.I01.
2. Rüeger, S.: Chapter 1 Data Analysis | Robust data analysis: An introduction to R, https://sinarueeger.github.io/robust-data-analysis-with-r/data-analysis.html, last accessed 2024/10/28.
3. Wickham, H.: You can't do data science in a GUI, https://www.youtube.com/watch?v=cpbtcsGE0OA, last accessed 2024/10/28.
4. Procida, D.: Diátaxis, https://diataxis.fr, last accessed 2024/10/28.
5. Schultz, L. Datadive Docs, https://docs.datadive.app, last accessed 2024/11/10.
6. Software Architecture | Software Engineering Institute, https://www.sei.cmu.edu/our-work/software-architecture/, last accessed 2024/10/28.
7. Rule, A: We Analyzed 1 Million Jupyter Notebooks — Now You Can Too, https://blog.jupyter.org/we-analyzed-1-million-jupyter-notebooks-now-you-can-too-guest-post-8116a964b536, last accessed 2024/10/28.
8. Landy, J: Notebook Research, https://github.com/jupyter/notebook-research, last accessed 2024/10/28.
9. Project Jupyter, https://jupyter.org, last accessed 2024/10/28.
10. The Jupyter Notebook Format, https://ipython.org/ipython-doc/3/notebook/nbformat.html, last accessed 2024/10/28.
11. Architecture - Jupyter Documentation, https://docs.jupyter.org/en/stable/projects/architecture/content-architecture.html, last accessed 2024/10/28.
12. ZeroMQ, https://zeromq.org, last accessed 2024/10/28.
13. Customizing User Environment - Zero to JupyterHub with Kubernetes documentation, https://z2jh.jupyter.org/en/latest/jupyterhub/customizing/user-environment.html#customize-an-existing-docker-image, last accessed 2024/11/04.
14. JupyterHub REST API – JupyterHub Documentation, https://jupyterhub.readthedocs.io/en/stable/reference/rest-api.html, last accessed 2024/11/04.
15. The REST API – Jupyter Server Documentation, https://jupyter-server.readthedocs.io/en/latest/developers/rest-api.html, last accessed 2024/11/04.

16. Interactive Jupyter Server API Documentation – Swagger, https://petstore.swagger.io/?url=https://raw.githubusercontent.com/jupyter/jupyter_server/master/jupyter_server/services/api/api.yaml, last accessed 2024/11/04.
17. Potvin, R., Levenberg, J.: Why Google stores billions of lines of code in a single repository. Communications of The ACM. 59, 7, 78–87 (2016). https://doi.org/10.1145/2854146.
18. Workspaces – Package Manager | Bun Docs, https://bun.sh/docs/install/workspaces, last accessed 2024/10/28.
19. Google TypeScript Style Guide, https://google.github.io/styleguide/tsguide.html, last accessed 2024/10/28.
20. Deno Style Guide, https://docs.deno.com/runtime/contributing/style_guide/#typescript, last accessed 2024/10/28.
21. TypeScript Deep Dive StyleGuide, https://basarat.gitbook.io/typescript/styleguide, last accessed 2024/10/28.
22. typescript-eslint, https://typescript-eslint.io, last accessed 2024/10/28.
23. Pocock, M: Intro To TypeScript Performance | Total TypeScript, https://www.totaltypescript.com/typescript-performance, last accessed 2024/10/28.
24. Performance • microsoft/TypeScript Wiki, https://github.com/microsoft/Typescript/wiki/Performance#preferring-interfaces-over-intersections, last accessed 2024/10/28.
25. Dorfmeister, D: Array Types in TypeScript, https://tkdodo.eu/blog/array-types-in-type-script, last accessed 2024/10/28.
26. Kamp, P.: Why Should I Care What Color the Bikeshed Is?, https://bikeshed.com, last accessed 2024/10/28.
27. Option Philosophy · Prettier, https://prettier.io/docs/en/option-philosophy, last accessed 2024/10/28.
28. Dodds, K.: Your code style does matter actually | Epic Web Dev, https://www.epicweb.dev/your-code-style-does-matter-actually, last accessed 2024/10/28.
29. Polanski, E: Introduction to Functional Programming using TypeScript and fp-ts, https://github.com/enricopolanski/functional-programming, last accessed 2024/10/28.

30. Suggestion: 'throws' clause and typed catch clause • issue #13219 • microsoft/TypeScript, https://github.com/microsoft/TypeScript/issues/13219, last accessed 2024/10/28.
31. Borla H: Announcing Swift 6, https://www.swift.org/blog/announcing-swift-6/, last accessed 2024/10/28.
32. Recoverable Errors with Result - The Rust Programming Language, https://doc.rust-lang.org/book/ch09-02-recoverable-errors-with-result.html, last accessed 2024/10/28.
33. Hudak, P., Peterson, J.; Fasel J.: About Monads - A Gentle Introduction to Haskell, https://www.haskell.org/tutorial/monads.html, last accessed 2024/10/28.
34. Nteifeh, H.: Functional Dependency Injection In Typescript, https://hugonteifeh.medium.com/functional-dependency-injection-in-typescript-4c2739326f57, last accessed 2024/10/28
35. Nault, A.: Dependency Inversion Principle in Functional TypeScript, https://alexnault.dev/dependency-inversion-principle-in-functional-typescript, last accessed 2024/10/28.
36. Factory Pattern, https://www.patterns.dev/vanilla/factory-pattern/, last accessed 2024/10/28.
37. Pocock, M.: Why I Don't Like Enums | Total TypeScript, https://www.totaltypescript.com/why-i-dont-like-typescript-enums, last accessed 2024/10/28.
38. 2024 Stack Overflow Developer Survey, https://survey.stackoverflow.co/2024, last accessed 2024/10/28.
39. State of JavaScript 2022, https://2022.stateofjs.com/en-US, last accessed 2024/10/28.
40. Octoverse: The state of open source and rise of AI in 2023 - The GitHub Blog, https://github.blog/news-insights/research/the-state-of-open-source-and-ai/, last accessed 2024/10/28.
41. WebAssembly, https://webassembly.org/docs/faq/#is-webassembly-trying-to-replace-javascript, last accessed 2024/10/28.
42. ECMAScript® 2025 Language Specification, https://tc39.es/ecma262/, last accessed 2024/10/28
43. TypeScript: JavaScript With Syntax For Types, https://www.typescriptlang.org, last accessed 2024/10/28.

44. Best of JS • Runtime projects, https://bestofjs.org/projects?tags=runtime, last accessed 2024/10/28.
45. Bun - A fast all-in-one JavaScript runtime, https://bun.sh, last accessed 2024/10/28
46. Kubernetes, https://kubernetes.io, last accessed 2024/10/28.
47. Hono - Web framework built on Web Standards, https://hono.dev, last accessed 2024/10/28
48. Getting Started - Hono, https://hono.dev/docs/getting-started/basic, last accessed 2024/10/28
49. Zod OpenAPI - Hono, https://hono.dev/examples/zod-openapi, last accessed 2024/10/28
50. Validation - Hono, https://hono.dev/docs/guides/validation#zod-validator-middleware, last accessed 2024/10/28
51. tursodatabase/libsql: libSQL is a fork of SQLite that is both Open Source, and Open Contributions, https://github.com/tursodatabase/libsql, last accessed 2024/10/28.
52. Welcome to Turso - Turso, https://docs.turso.tech, last accessed 2024/10/28.
53. Kysely, https://kysely.dev, last accessed 2024/10/28.
54. RobinBlomberg/kysely-codegen: Generate Kysely type definitions from your database, https://github.com/RobinBlomberg/kysely-codegen, last accessed 2024/10/28.
55. React, https://react.dev, last accessed 2024/10/28.
56. TanStack Router, https://tanstack.com/router/latest, last accessed 2024/10/28.
57. TanStack Query, https://tanstack.com/query/latest, last accessed 2024/10/28.
58. Vite | Next Generation Frontend Tooling, https://vite.dev, last accessed 2024/10/28.
59. Vitest | Next Generation Testing Framework, https://vitest.dev, last accessed 2024/10/28.
60. Tailwind CSS - Rapidly build modern websites without ever leaving your HTML, https://tailwindcss.com, last accessed 2024/10/28.
61. State of CSS 2024, https://2024.stateofcss.com/, last accessed 2024/10/28.
62. Astro, https://astro.build, last accessed 2024/10/28.
63. Starlight with Astro® Build documentation sites, https://starlight.astro.build, last accessed 2024/10/28.

64. colinhacks/zod: TypeScript-first schema validation with static type inference, https://github.com/colinhacks/zod, last accessed 2024/10/28.
65. supermacro/neverthrow: Type-Safe Errors for JS & TypeScript, https://github.com/supermacro/neverthrow, last accessed 2024/10/28.
66. Radix Primitives, https://www.radix-ui.com/primitives, last accessed 2024/10/28.
67. shadcn/ui, https://ui.shadcn.com, last accessed 2024/10/28.
68. Find and fix problems in your JavaScript code - ESLint - Pluggable JavaScript Linter, https://eslint.org, last accessed 2024/10/28.
69. Prettier, https://prettier.io, last accessed 2024/10/28.
70. Introduction | Turborepo, https://turbo.build/repo/docs, last accessed 2024/10/28.
71. Resend, https://resend.com, last accessed 2024/10/28.
72. Zero to JupyterHub with Kubernetes - Zero to JupyterHub with Kubernetes documentation, https://z2jh.jupyter.org/, last accessed 2024/11/04.
73. Visual Studio Code - Code Editing. Redefined, https://code.visualstudio.com, last accessed 2024/10/28.
74. Authenticators – JupyterHub Documentation, https://jupyterhub.readthedocs.io/en/latest/reference/authenticators.html#how-to-write-a-custom-authenticator, last accessed 2024/11/04.
75. Contents API - Jupyter Server documentation, https://jupyter-server.readthedocs.io/en/latest/developers/contents.html#contents-api, last accessed 2024/10/28.