Thoth: A Domain-Specific Language for Multitier Web Development

Submitted in conformity with the requirements for the degree of
MEng in Computer Science and Software Engineering

Abdullah Elsayed
Supervised by: Dr. Vincent Rahli

School of Computer Science
University of Birmingham

April 2023

I, Abdullah Elsayed confirm that the work presented in this dissertation is my own. Where information has been derived from other sources, I confirm that this has been indicated in the dissertation.

Abstract

Multitier programming enables the development of the typical 3-tiers (database, server, and client) of a web application in a single compilation unit using a multitier language. This approach makes developing web applications easier than the current approach, which requires combining multiple tools and programming languages. There have been many multitier programming languages proposed in the literature, but they have failed to gain the attention of the community because they usually lack support for the tools required to build web applications.

We advocate that multitier programming languages should be interoperable with existing general-purpose languages, enabling developers to benefit from the rich ecosystem of these languages. The proposed language is designed to address the impedance mismatch problem that can happen in the 3-tier architecture as well as reduce the amount of boilerplate code required in web app development. It offers a set of declarations that can be used to develop all parts of web applications and abstracts the low-level details of web development. Additionally, it is designed to be interoperable with TypeScript to address the problem of adoption that faced previous multitier languages. This enables developers to benefit from the rich ecosystem of JavaScript/TypeScript. The language also compiles to human-readable code using existing tech stacks, giving developers the freedom to take the compiled code and further develop their apps when the language becomes limiting.

The language is evaluated through a series of experiments that measure its effectiveness in reducing development effort, eliminating the impedance mismatch problem, and reducing the learning curve. The results indicate that the language can significantly reduce the amount of code required to build a web app while improving the quality of the code and that it is easy to learn.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

We propose a new multitier domain-specific language (DSL) called **Thoth** that enables developers to build the database, server, and client tiers of web applications as a single, mono-linguistic program. Thoth enables developers to create web apps without writing excessive boilerplate code by offering a set of declarations that abstract most of the low-level details of web development. Multitier development is a better approach for developing web applications because with the currently available tools, developers have to build the different parts of the application using different tools and programming languages. This separation can cause a problem best known as the *impedance mismatch* problem, as we are going to explain in section 1.1.1. The declarative approach that our language implements has never been implemented by other multitier languages before, as we will see later in this chapter. Using the DSL, developers can define data models, queries, and interactive user interfaces, as well as implement common web features like authentication and authorization seamlessly.

In addition, Thoth compiles to real-time applications by default using server-sent events (SSE). It compiles to a NodeJS server and a ReactJS client, but it is designed to be framework-agnostic. This means that the compiler can be easily modified to compile to other frameworks. Thoth is also interoperable with TypeScript, which allows developers to benefit from the JavaScript/TypeScript ecosystem. The declarative nature of the language can be limiting, which is why we focused on creating a way for developers to easily customize the DSL declarations. Additionally, we wanted to enable them to take the compiled application and further develop it without using the DSL. That is precisely why we designed the language to compile to easily understandable human-readable code. We will discuss how we achieved these goals in greater detail in the following chapters.

1.1 Background

Modern web apps are typically built using the 3-tier architecture. In this architecture, the application is divided into three layers, each with its own responsibility and functionality (Figure 1.1). The client tier is the first tier of the architecture, and it is in charge of rendering the user interface and handling user interactions. This tier includes the client application, which runs on the user's browser and is typically built with HTML, CSS, and JavaScript. The client app sends HTTP requests to the backend API and receives responses in JSON, XML, or HTML format. The logic tier is the architecture's second tier. This tier consists of a backend API, which is typically written in a programming language such as Java, Python, Go, or others. It is in charge of processing client requests and responding to them. It is also in charge of handling data validation, authentication, and permissions, as well as communicating with the database to perform create, read, update, and delete (CRUD) operations on the data. The database tier is the third tier of the architecture, and it is in charge of storing and managing data. This tier consists of a database running on the same or a separate machine from the server. It is created with a database management system such as MySQL, PostgreSQL, or MongoDB, which handles data storage, retrieval, and manipulation while ensuring data consistency and integrity.

In summary, the lifecycle of most web apps works as follows: whenever a user interacts with the client app, the app sends an HTTP request to the backend API. The backend API receives and processes the request, then sends a response back to the client app. Finally, the client app renders the response on the screen for the user.

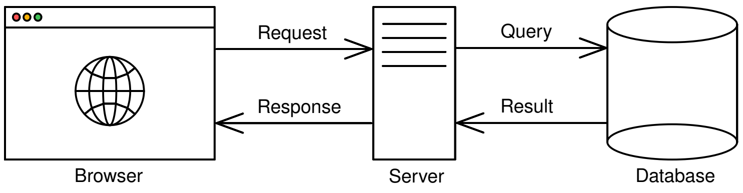


Figure 1.1: A diagram representing the 3-tier architecture.

1.1.1 Impedance Mismatch Problem

However, the separation in layers that the 3-tier architecture provides can help make the development of full-stack apps more modular because each layer can be developed and maintained independently. This can cause a problem known as *impedance mismatch*. In general, impedance mismatch happens when two components in a system have different data models, data structures, or data representations. This can lead to communication problems, which can cause unexpected errors in the system and prevent it from functioning correctly.

For instance, suppose the client app has an HTML form that sends the wrong data representation to the server. As shown in Figure 1.2a, the client app sends the age to the server as a string, but the server expects the age as an integer. In this case, the server will fail to handle the HTTP request coming from the client, leading to unexpected results. Similarly, this can happen between the server and the database when the server implements the wrong data models for the tables in the database. As shown in Figure 1.2b, the server is trying to select a field called `email` from the `user` table, but the user table in the database does not have a field called `email`.

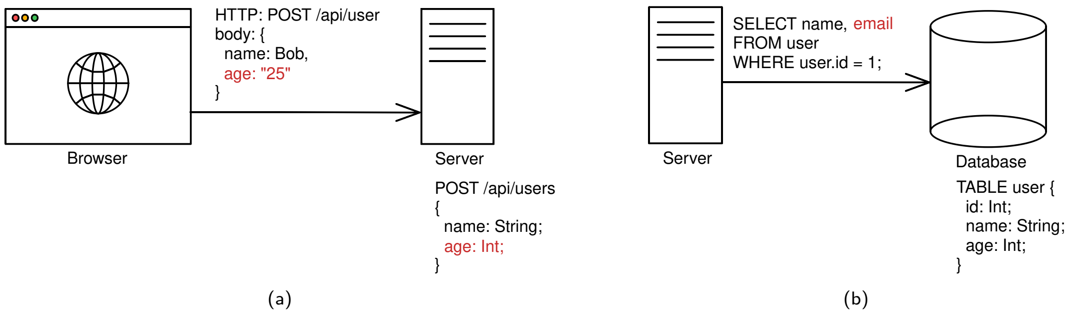


Figure 1.2: Impedance mismatch problem.

1.1.2 Boilerplate Code Problem

Building full-stack apps requires too much boilerplate code, which makes the development time-consuming. Boilerplate code in full-stack web applications refers to the repetitive code that must be written. This code is often necessary for the app to function correctly but does not directly contribute to the application's unique features or functionality. This includes code for writing input validation schemas, setting up the middlewares that are used for data validation, etc. Moreover, most modern web apps usually have common features like authentication and permissions, which can be tedious to setup using the currently available tools. Here are some examples of common boilerplate code:

1- **Validation schemes**: Validation schemas can be used by validation middlewares to validate if the client app sent the right data or not. Listing 1.1 shows a schema that validates that clients sent a JSON object with fields `username` and `password` of type string. When the client fails to send any of the required fields or sends the incorrect data types, this schema will throw an error. The code uses a TypeScript validation library called Zod[1].

```
/\*_ An example in NodeJS/ExpressJS and Zod _/
import { z } from 'zod';

export const signup = z.object({
data: z.object({
username: z.string({
required_error: '`username` is required',
}),
password: z.string({
required_error: '`password` is required',
}),
}),
});
```

Listing 1.1: Validation schema example.

2- **Middlewares**: Middlewares are used to perform extra operations on the request before handing it to the function that is responsible for handling it. Middlewares are usually used to do data validation or check user permissions. The code in Listing 1.2 shows how middlewares can be used for data validation.

```
/\*_ An example in NodeJS/ExpressJS _/
const express = require('express');
const app = express();

/\*\*

- A function that uses the validation schema
- and validates the body of the request
  _/
  function validationMiddleware(req, res, next) { /_ body \*/ }

/\*\*

- A function that handles the requests
  _/
  function signUpUser (req, res, next) { /_ body \*/ }

/\*_ Route _/
app.post('/api/auth/signup', validationMiddleware, signUpUser);
```

Listing 1.2: Middlewares example.

3- **Simple CRUD**: Simple CRUD queries require routing, request handling, and validation schema. Instead of just writing the database query, developers will have to implement that for most API endpoints. For example, if the API endpoint just implements a simple read query that reads all records from a table in the database, it has to be implemented with all the boilerplate around it to be functional.

4- **Implementing features twice**: Some features have to be implemented twice Like authentication and validation. For example, if some content in the app requires authentication to be viewed by the users, the authentication part of the app has to be implemented once on the client-side and again on the server-side. Also, validation schemas have to be implemented twice for validating forms on the client and validating the data sent by the client on the server.

1.2 Related Work

There are existing solutions in the literature that try to solve the problems that we have introduced. In this section, we are going to talk about the approaches of each one and their drawbacks.

1.2.1 Multitier Programming

In recent years, there has been a growing interest in multitier programming in literature. Several languages have been proposed to allow for the development of full-stack web applications in a single language without the client-server distinction. Multitier languages are beneficial for developing web apps because developers can write a complete web app without thinking about the client-server distinction, and the compiler splits the code as needed and takes care of all the communication. Here are some of the languages that have introduced novel approaches to developing full-stack web apps:

Links

Links[2] is a statically-typed functional language that offers developers a solution to create web apps in a single language. It achieves that by requiring the developers to annotate their code with “client” or “server” annotations on functions levels so that the compiler can split the code. Any app built using Links is compiled to an OCaml server, a client in JavaScript, and database queries in SQL. Although Links provides a type-safe web programming model, which reduces the risk of errors and vulnerabilities in the application, it still has several problems that prevent it from being widely adopted. It is a functional programming language and has unfamiliar syntax, which can make it difficult to learn for developers who are not familiar with this paradigm.

Opa

Similar to Links, Opa[3] is a statically-typed functional language that offers the same features as Links and also requires the code to be annotated. Opa apps are compiled into a NodeJS server, a JavaScript client app, and a MongoDB database. It automates many aspects of modern web application programming, such as Ajax/Comet client-server communication, event-driven and non-blocking programming models. Opa did not have the same issues as Links because it used syntax similar to JavaScript, which many developers are familiar with. Moreover, Opa allows developers to reuse existing JavaScript libraries by enabling them to interact with external JavaScript code. Additionally, Opa’s standard library includes the popular UI library jQuery, further enhancing its capabilities for web application development. This made it easy for developers to learn Opa and adopt it.

Hop

Unlike Links and Opa, Hop[4] is a dynamically-type language. It is one of the first multitier languages introduced to solve the problem of developing full-stack apps in a single code that runs on the client and the server. The language has native support for HTML, server-side web workers, and WebSockets. Unlike Links and Opa, the language syntax is similar to JavaScript and allows developers to specify server-client annotations on an individual expression level. Developers use the `service` keyword to define services, which are essentially JavaScript functions that are automatically invoked when an HTTP request is received. Unlike Links and Opa, developers can use the `~{` marker within the service to switch from server-side to client-side context. For example, in the code snippet in Listing 1.3, `alert("world")` will run on the client while the rest will run on the server.

```
service hello() {
return (

<html>
<div onclick=~{ alert("world") }>hello</div>
</html>
);
}
```

Listing 1.3: Hop example.

1.2.2 No-boilerplate Programming

Frameworks are typically used to build each layer of a web app. To build client apps, for example, developers typically use JavaScript UI frameworks such as ReactJS or Angular. Similarly, they build backends with frameworks such as Django, Flask, or ExpressJS. Using any of the tools mentioned reduces the amount of boilerplate code that must be written. However, there is still boilerplate code required to be written, such as writing configuration files for both the client and server apps, as well as extra code such as the examples mentioned previously.

Wasp

Wasp[5] appeared in 2020, offering a novel approach to developing full-stack apps without boilerplate code. Its goal is to enable developers to describe the common features required by most web apps and then write the rest in ReactJS for the client app, NodeJS for the server app, and Prisma for the database. Wasp accomplishes this through the use of declarations. Wasp declarations allow developers to write what they want rather than how they want it. Wasp is only a boilerplate prevention solution; it is not a multitier language like Links or Opa.

Chapter 2

Analysis and Specification

As discussed in the previous chapter, there are several existing tools in the literature designed to support multitier programming for full-stack apps and to address the issue of boilerplate code. Some of the previously introduced multitier programming languages faced challenges that hindered their widespread adoption. For instance, languages such as Links and Hop lacked interoperability with existing general-purpose languages, which made them lack support for tools required for web app development while Opa allowed developers to access JavaScript libraries. Languages like Links were built using a syntax and programming paradigm unfamiliar to most developers, making them difficult to learn and adopt.

As shown in Figure 2.1, Opa was the most preferred language compared to the other two. Our approach aims to create a DSL that is interoperable with TypeScript and enables developers to access JavaScript libraries, like Opa. Additionally, it will address the problem of boilerplate code using the same approach as Wasp. In this chapter, we will outline the requirements for a multitier language for developing web apps that might be appealing to developers.

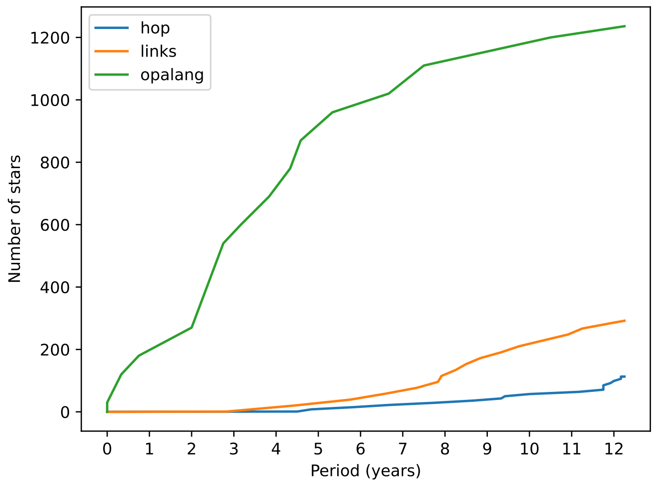


Figure 2.1: The stars of each GitHub repository over time.

R1. Support multitier programming

The DSL must enable developers to write web applications in a single program without thinking about the client-server distinction. Data models, database queries, and user interfaces should all be defined in the same compilation unit. Although developers can use server resources directly on the client, the syntax of the language should still define a clear distinction between client and server resources so that no secrets are accidentally revealed. The language will be compiled and checked for errors across all tiers at compile time. This will prevent problems like *impedance mismatch* and thus limit runtime errors.

R2. Allow developing full-stack apps declaratively

The DSL must allow developers to specify application features using declarative syntax. Declarative programming allows developers to express the desired outcome of an operation or process rather than specifying how to achieve that outcome. For common full-stack web application functionality such as forms, routing, authentication, and permissions, the DSL must provide high-level abstractions and components. The declarative syntax must be intuitive and simple to learn, allowing developers to build applications quickly without wasting time and effort on implementation details. This declarative approach was inspired by the Wasp model, which demonstrated great potential for avoiding boilerplate code.

The DSL will help increase productivity by eliminating the need for boilerplate code and improving the maintainability of the application code by allowing developers to develop full-stack web applications declaratively. Furthermore, it will eliminate the need for previous multitier languages' client-server annotation method, which was used to allow the compiler to distinguish between the client and the server. The annotation method is typically error-prone and makes debugging the code difficult. By using declarations, the compiler will be able to differentiate between client and server code, the codebase will remain readable for developers, and the developer experience will be enhanced.

R3. Compile to human-readable and type-safe TypeScript code using existing tech stacks

The DSL must provide a compiler that can generate human-readable and type-safe TypeScript code. This means that the compiler must be able to translate the DSL's declarations and high-level abstractions into low-level TypeScript code that developers can easily understand and modify. To compile well-structured apps, the DSL must make use of existing tech stacks. The generator of the compiler will only compile to a NodeJS server and a ReactJS client because this is the most used stack in the world according to the "Stack Overflow Developer Survey 2022"[6]. Even though the DSL must compile to NodeJS and ReactJS, it should be framework-agnostic. This means that the compiler should be able to compile to any currently available tech stacks.

By compiling existing tech stacks to human-readable and type-safe code, developers can take the compiled code and further develop it without using the DSL. Because the declarative approach with high-level abstractions that we are attempting to achieve can sometimes be limiting, it is necessary to provide developers with the ability to easily opt out of developing their app using the DSL.

R4. Real-time by default

The compiled app must include real-time functionality by default, allowing developers to create applications that can update and display data in real-time without writing additional code or configurations. This means that the application must receive and display data and state updates as they occur, without the need for a manual refresh of the page. This functionality can be accomplished through the use of technologies such as WebSockets, long/short polling, or SSE.

By making the compiled app have real-time functionality by default, the DSL will increase the interactivity and responsiveness of the app, providing a more seamless and engaging user experience. This is also beneficial for applications that require frequent updates, such as chat apps and collaborative tools. Furthermore, it will save developers time and effort when implementing real-time functionality and ensure that the app meets the expectations of users who expect highly interactive apps.

R5. Allow interoperability with TypeScript

The DSL must be interoperability with TypeScript, a popular statically-typed superset of JavaScript. This will enable developers to write custom code and extend the DSL's functionality. Because of the interoperability with TypeScript, developers will be able to take advantage of the rich ecosystem of third-party libraries and tools available in the JavaScript/TypeScript ecosystem. The JavaScript/TypeScript ecosystem is one of the largest in the world of web development, with thousands of libraries and tools available to help developers build web applications. By making the DSL interoperable with TypeScript it should avoid the lack of community support that affected previous multitier languages.

Chapter 3

Key Features and Syntax

In this chapter, we will provide a comprehensive introduction to the DSL by explaining its key features and syntax. The main goal of this chapter is to give a thorough understanding of the language's declarations and expressions. The DSL's syntax is designed around a few declarations. In each section in this chapter, we are going to explain how to use these declarations to build web applications using Thoth. For information on the language grammar defined in Backus–Naur form (BNF), please refer to Appendix 1.

3.1 App Configurations

The `app` declaration serves as the project's starting point and is used to define global configurations such as the app's title and authentication configuration. Every app built with the DSL must be declared, and there can only be one `app` declaration. The `app` declaration takes a JSON-like object as a value. In the `app` declaration body, only the `title` field is required. The app's `title` will appear in the browser tab. We will talk in more detail about authentication in a later section. Apps can be declared as shown in the code in Listing 3.1.

```
app ExampleApp {
  title: "Example App"
}
```

Listing 3.1: App config example.

3.2 Data Models

Data models represent entities in the application domain and map to database tables. Models are declared using the `model` declaration. Every model must be declared with a name following the PascalCase convention. Each model must define several fields. For example, a blogging platform can define `Post` data model as shown in Listing 3.2.

3.2.1 Model Fields

Model fields are like columns in a database table. They define what kind of data can be stored in the table. Each field has a name, a type, an optional type modifier, and an optional attribute. The field name is a string that identifies the field within the model. The name must start with a letter and can be followed by any combination of letters, numbers, or underscores. It is common to name fields using the camelCase convention.

When defining model fields, it is necessary to specify the type of each field when defining it. There are two main categories of types: scalar types and user-defined types. The DSL has four primary scalar types: `Int`, `String`, `Boolean`, and `DateTime`. User-defined types refer to data models that are created by the user. These user-defined types can be used as types for fields to specify relations between different models in the application. In the next section, we will elaborate on the topic of relations in greater detail.

The type of field can be modified by appending either of two modifiers: `[]` to make it a list or `?` to make it optional. Optional lists are not supported, so type modifiers cannot be combined.

Attributes are also optional and can be used to provide additional information about a field. For example, you can specify that a field is unique (i.e., it cannot have duplicate values). All attributes start with the special character `@`. Fields can have one or more of the following attributes:

- **@id**: The `@id` attribute defines a unique field that can be used to identify individual model records. A model can only have one ID field because a field that implements the `@id` attribute is similar to a `PRIMARY_KEY` field in relational databases. The ID field must be of `Int` type.
- **@unique**: The `@unique` attribute can be used to make a field have unique values. For example, in a blogging platform, each post must have a unique slug, which is a human-readable, unique identifier used to identify a post instead of id.
- **@default**: Scalar fields can be created with default values using the `@default` attribute. The `@default` attribute takes the default value as an argument. The default value must be a literal value of the same type as the field type, such as `5` (`Int`), `"Hello"` (`String`), `false` (`Boolean`), or `Now` (`DateTime`). `Now` is a keyword that can be used to set a timestamp of the time when a record is created.
- **@updatedAt**: The `@updatedAt` attribute can be used with fields of type `DateTime` to automatically store the time when a record was last updated.
- **@relation**: This attribute is used to define relation fields. In the next section, we will elaborate on the topic of relations in greater detail.

The code in Listing 3.2 shows how fields can be defined using the concepts explained above. The syntax for defining fields is similar to Prisma Schema Language[7]. First, the field name is declared, which is then followed by the type, type modifier, and any number attributes. Each field must be declared on a new line, and the field name, type, and attributes must be separated using a space. The example implements a `Post` model that has an ID field, uses all the scalar types in the language, makes use of type modifiers, and shows how field attributes can be used.

```
model Post {
  id        Int      @id // an id field
  title     String // a `String` field that does not have any attributes
  slug      String?  @unique // a unique optional field of type `String`
  content   String
  published Boolean  @default(false) // a field with `false` as a default value
  createdAt DateTime @default(Now) // a field with a `Now` as default value
  updatedAt DateTime @updatedAt @ignore // a `DateTime` field that gets updates whenever a post
       record is updated and is ignored whenever posts are queried
}
```

Listing 3.2: Model example.

3.2.2 Relations

As mentioned before, models can define relations between each other. Relations are defined between models using user-defined types and the `@relations` attribute. The relation must be defined in both related models or the compiler will throw an error. The `@relation` attribute takes two arguments: the first is called the relation field, and the second is called the reference field. There are three types of relations: one-to-one relations, one-to-many relations, and many-to-many relations.

One-to-One Relations

One-to-one relations refer to relations where at most one record can be connected on both sides of the relation. For example, in the code in Listing 3.3 each user is associated with zero or one profile through the relation field `profile` in the `User` model. The `user` field in the `Profile` model defines a relation with the `User` model by referencing the field `id` in the `User` model. The `userId` relation field represents the foreign key. In this case, the relation field must implement the `@unique` attribute to make sure that every user is associated with only one profile.

```
 1 model User {
 2   id      Int      @id
 3   profile Profile?
 4 }
 5
 6 model Profile {
 7   id     Int  @id
 8   user   User @relation(userId, id)
 9   userId Int  @unique // relation field (used in the `@relation` attribute in line 8)
10 }
```

Listing 3.3: One-to-one relation example.

One-to-Many Relations

One-to-many relations refer to relations where one record on one side of the relation can be connected to zero or more records on the other side. As shown in the code in Listing 3.4, each user has zero or many posts, and each post has only one user. Similar to the previous example, the `User` model defines a relation with the `Post` model via the `posts` relation field. Then, the `Post` model references the `id` field in the `User` model using the `@relation` attribute. Unlike one-to-one relations, the relation field `userId` in the model `Post` must not implement the `@unique` attribute because a user can have more than one post.

```
model User {
  id    Int    @id
  posts Post[]
}

model Post {
  id     Int  @id
  user   User @relation(userId, id)
  userId Int
}
```

Listing 3.4: One-to-many relation example.

Many-to-Many Relations

Many-to-many relations refer to relations where zero or more records on one side of the relation are connected to zero or more records on the other side. Many-to-many relations are easy to implement by defining that each model has a list of the other model. As shown in the example in Listing 3.5, The `Post` model defines a relation field `tags` with the `Tag` model. Similarly, the model `Tag` defines a relation field `posts` with the `Post` model.

```
model Post {
  id    Int   @id
  tags  Tag[]
}

model Tag {
  id    Int    @id
  posts Post[]
}
```

Listing 3.5: Many-to-many relation example

3.3 Queries

The declaration of queries is a fundamental feature of the DSL that is designed to facilitate the creation of queries on data models. It offers five distinct types of queries, including `FindMany`, `FindUnique`, `Create`, `Update`, and `Delete`. Queries are defined using the `query` declaration, which is one of the declarations in the DSL that must take a type and a JSON-like object as a body. The body can implement one or more of the following entries: `where`, `search`, or `data`, based on the query type. Moreover, they must define the data model to query. This can be done using the query attribute `@model`. Furthermore, they run on the server only. Generally, query names should follow the camelCase convention.

The `where` entry is used to specify a unique field in the data model, and it is used to select an individual record using the unique field. In contrast, the `search` entry is used to filter a list of records based on the list of fields. The `data` entry is used to specify the fields that a `Create` or `Update` query can use to create a new record or update an existing one.

FindMany Queries

`FindMany` queries are used to read a list of records from a data model. Any `FindMany` query must implement a `search` entry that takes a list of fields that can be used to filter the records in the database. `FindMany` queries cannot implement either the `where` or the `data` entries. For example, for the `Post` model shown in Listing 3.2, we can implement a `FindMany` query that returns a list of posts or filter them based on the `published` field, as shown below:

```
@model(Post)
query<FindMany> getPosts {
  search: [ published ]
}
```

Listing 3.6: FindMany query example.

FindUnique Queries

Individual records can be read using `FindUnique` queries. `FindUnique` queries must implement the `where`, and it must take a unique field, for the query to identify the record. For instance, to get a post by id, this query can be defined as follows:

```
@model(Post)
query<FindUnique> getPostById {
  where: id
}
```

Listing 3.7: FindUnique query example.

Create Queries

Create queries can only implement the `data` entry, which can be used to specify the fields needed to create a new post record. The `data` entry takes two entries: `fields` a required entry that specifies non-relation fields and `relationFields` an optional entry that specifies how models should be linked with each other. Create queries must specify all the required fields in a model. For instance, consider the `Post` model in the example below, the create query for this model must specify the `title`, `content`, and `user` fields because they are all required. Any optional or list fields do not have to be specified. As well as fields that have `@default` or `@updatedAt` attributes, because records can be created using the default values.

The `Create` query shown below specifies that the query can create a post record using the `title` and `content` fields in the `fields` entry. While the `relationFields` specifies how a post record can be linked to a user using the `connect-with` expression. In this case, the `user` field is connected to the `id` field of a user record using `userId` field in a post record as a foreign key field without specifying them.

```
@model(Post)
query<Create> createPost {
  data: {
    fields: [ title, content ],
    relationFields: {
      user: connect id with userId
    }
  }
}
```

Listing 3.8: Create query example.

Update Queries

A record can be updated by defining an update query. Update queries must define the `where` and `data` entries. Similar to `FindUnique` queries, the `where` entry must specify a unique field, for the query to be able to find the record that needs to be updated. Where the `data` entry takes the field that can be updated. The example below defines an update query that can update the `title`, `content`, or `published` fields of a post record.

```
@model(Post)
query<Update> updatePostById {
  where: id,
  data: {
    fields: [ title, content, published ]
  }
}
```

Listing 3.9: Update query example

Delete Queries

To define a delete query, the query must only implement the `where` entry. The body of the delete query is similar to `FindUnique` queries.

```
@model(Post)
query<Delete> deletePostById {
  where: id
}
```

Listing 3.10: Delete query example.

3.4 UI Components

User interfaces can be built using the `component` declaration. Components can be declared with a type, or without declaring a general UI component. There are several types of components, including fetch components (FindMany, FindUnique) and action components (Create, Update, Delete). Fetch components are used to get data from the server using a query, to represent it to the user. While action components are used to mutate data on the server. All component declarations run on the client. Like models, UI component names must follow the PascalCase convention.

3.4.1 Render Expressions

UI structures can be built using the render expression which, takes XML-like structures. The DSL provides a modified version of HTML called XRA that can be mixed with for-expressions, if-expressions, variables, literals, or user-defined UI components. Elements within the render expression can be styled using any CSS classes provided by the WindiCSS[8], which is a utility-first CSS framework integrated natively in the DSL. Within render expressions, literals and variables can be rendered using the `{ ... }` notation as follows:

```
render(<div>{ "Hello, world" }</div>) // renders a string
render(<div>{ 42069 }</div>) // renders a number
render(<div>{ user.firstName }</div>) // renders a variable
```

Listing 3.11: Rendering literals and variables example.

if-expressions can be declared using the following syntax:

```
// if-expression
render(
  <div>
    [% if condition %]
      { "Condition is true" }
    [% endif %]
  </div>
)

// if-else-expression
render(
  <div>
    [% if condition %]
      { "Condition is true" }
    [% else %]
      { "Condition is false" }
    [% endif %]
  </div>
)
```

Listing 3.12: If-expression example.

Moreover, for-expressions can be declared using a similar syntax as if-expression:

```
render(
  <div>
    [% for element in list %]
      { element }
    [% endif %]
  </div>
)
```

Listing 3.13: For-expression example

3.4.2 General Components

Typically, general components are used to build stateless UI components. This is useful to make building UIs more modular and makes maintaining UI components easier. The example below shows a component that takes an argument `post` of type `Post` as an argument and then presents the `title` and `content` with a text of color gray using the CSS classes provided by WindiCSS.

```
component PostDetailComponent(post: Post) {
  render(
    <div className="text-gray-800">
      <div>{ post.title }</div>
      <div>{ post.content }</div>
    <div>
  )
}
```

Listing 3.14: General component example.

3.4.3 Fetch Components

There are two types of components that fetch data using a read query. These are `FindUnique` and `FindMany` components. Fetch components must implement a JSON-like body that takes 4 required entries: `findQuery`, `onError`, `onLoading`, and `onSuccess`. The `findQuery` entry takes a query of type `FindMany` or `FindUnique` depending on the component type. This query is used ot fetch the data from the server. While the entries must take a render expression with a UI structure to view for the user on each case. For example, we can implement a `FindMany` component that presents a list of posts to the user using the query shown in Listing 3.6, as follows:

```
 1  component<FindMany> PublishedPostsList {
 2    findQuery: getPosts({
 3      search: {
 4        published: true
 5      }
 6    }) as posts,
 7    onError: render(<div>{ "Error getting posts" }</div>),
 8    onLoading: render(<div>{ "Loading posts..." }</div>),
 9    onSuccess: render(
10      <div>
11        [% for post in posts %]
12          <PostDetailComponent post={ post } />
13        [% endfor %]
14      </div>
15    )
16  }
```

Listing 3.15: FindMany component example.

The above example defines a component that uses the `getPosts` query to get a list of published posts and then saves the result in the variable `posts`. When a `FindMany` query is used, it can take an optional JSON object to set values for the search fields defined in the query. In each case, the component will render the appropriate UI structure for the user. As shown in line 12, the component makes use of the component implemented in Listing 3.14 to show the post details.

3.4.4 Action Components

Action components provide a way for developers to create user interfaces that enable users to mutate data by performing actions, such as creating, updating, and deleting. `Create` components, for example, are used to create forms that enable users to add new data to a particular model. These components utilize a query to define the fields that the form should contain. `Update` components, on the other hand, are used to create forms that enable users to update existing data in a model. These components are similar to `Create` components but include pre-populated values for the fields that the user is updating. Finally, `Delete` components are used to create action buttons that enable users to delete a record from a particular model.

3.4.4.1 Create and Update Forms

`Create` and `Update` components has several entries that define their behavior and structure. Firstly, the `actionQuery` entry specifies a query that will be executed when the form is submitted. This query is responsible for creating a new record in the database with the data submitted in the form or updated an existing one. Secondly, the `formInputs` entry is used to define the input fields in the form. Each input field is defined by a name and an input type. Additional entries can also be added to the input field definition, such as styles, placeholders, and default values. The `label` entry can also be used to define a label for each input field, improving the accessibility and usability of the form. Thirdly, the `formButton` entry is used to define the button that will submit the form. This entry allows developers to customize the text displayed on the button, as well as any styles that should be applied to the button. Finally, the `globalStyle` entry is used to define the global styles that will be applied to the entire form. This entry allows developers to customize the appearance of the form, such as the background color and font styles.

The DSL provides a range of input types that developers can use to create type-safe forms for their web applications. The available input types include `TextInput`, `EmailInput`, `PasswordInput`, `NumberInput`, `RelationInput`, `DateTimeInput`, `DateInput`, and `CheckboxInput`. The `TextInput` input type is used to capture textual data, while the `EmailInput` input type is specifically designed to capture email addresses. The `PasswordInput` input type is used to capture passwords, with the input field automatically masking the characters entered. These three input types can be implemented with fields of type `String`. The `NumberInput` input type is used to capture numerical data, such as age or quantity, and can be used with fields of type `Int` only. The `RelationInput` input type is used to capture relational data, such as linking a user to a post they authored. The `DateTimeInput` input type is used to capture both a date and time, while the `DateInput` input type is used to capture only a date. These two input types can be used with `DateTime` fields. Finally, the `CheckboxInput` input type is used to capture `Boolean` data, such as selecting a checkbox to indicate agreement to terms and conditions.

The code shown in Listing 3.16 implements a `Create` component, which uses the `createPost` query implemented in Listing 3.8 as its action query. Moreover, the component implements form inputs, which are inputs specified in the `createPost` query. It implements two text inputs for the `title` and `content` fields. Additionally, it implements an input `user` of type `RelationInput`, which specifies that the submitted post should be linked to the currently logged-in user. The value for this input is pre-populated by default and hidden from the user. The form cannot implement fields that are not specified in the `createPost` query. It also must implement all the fields specified by the query. In the end, the component defines a form button, which is the button used to submit the data. The provided example uses the `globalStyle` entry to define global styles for the form elements. In the `globalStyle`, the `form` entry defines the background color of the form, while the `inputContainer` and `input` entries define the background color of the input container and the input field, respectively. The `inputLabel` entry defines the style of the label associated with the input field, and the `inputError` entry defines the style of the error message displayed if the input is invalid. In addition to the global styles, developers can also define styles for individual input fields using the `style` entry. In the example provided, the `style` entry is used to override the global styles to set a different background color and text color to elements in the `title` input field. The styling for this example is represented by the diagram in Figure 3.1.

```
component<Create> PostCreateForm {
  actionQuery: createPost(),
  globalStyle: {
    form: "bg-green-500",
    inputContainer: "bg-blue-500",
    input: "bg-red-500",
    inputLabel: "text-black",
    inputError: "text-red-500"
  },
  formInputs: {
    title: {
      style: "bg-white",
      label: {
        style: "text-red-500",
        name: "Post Title"
      },
      input: {
        type: TextInput,
        placeholder: "Enter post title",
        style: "bg-white text-red-500"
      }
    },
    content: {
      label: {
        name: "Post Content",
      },
      input: {
        type: TextInput,
        placeholder: "Enter post content"
      }
    },
    user: {
      input: {
        type: RelationInput,
        isVisible: false,
        defaultValue: connect id with LoggedInUser.id
      }
    }
  },
  formButton: {
    style: "bg-yellow-500",
    name: "Create"
  }
}
```

Listing 3.16: Create form example.

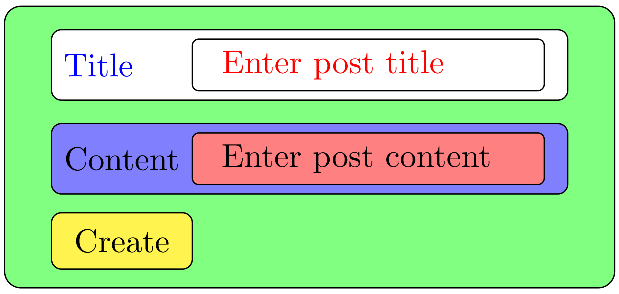


Figure 3.1: A diagram representing how form styling works.

3.4.4.2 Delete Buttons

`Delete` buttons are created like `Create` and `Update` components but `Delete` buttons only implement `actionQuery` and `formButton` entries. The example below uses the `deletePostById` query implemented in Listing 3.10 and passes the value of the argument `id` to the `where` argument of the query.

```
component<Delete> PostDeleteButton(id: Int) {
  actionQuery: deletePostById({
    where: id
  }),
  formButton: {
    style: "bg-yellow-500",
    name: "Delete"
  }
}
```

Listing 3.17: Delete button example.

3.5 Pages

Pages in the DSL are defined by `page` declarations. Pages are created by composing one or more components and they define the layout and structure of the page. When a user navigates to a specific route, the corresponding page is displayed. Page names must follow the PascalCase convention. Every page must implement an attribute called `@route`, which takes a string to define the route for the page. In the example below, the page `Home` renders the two components `PublishedPostsList` and `PostCreateForm` implemented in Listing 3.15 and Listing 3.16 consecutively. This page can be accessed by navigating to the root route of the app.

```
@route("/")
page Home {
  render(
    <PublishedPostsList />
    <PostCreateForm />
  )
}
```

Listing 3.18: Page example.

3.6 Authentication and Permissions

Authentication is a common feature in modern web apps and it is important to provide developers the ability to implement this feature in their apps in the simplest way possible. To implement authentication in the app, developers have to create a data model that represents users and then define in the app configuration that the app should have authentication using this data model. The user model must define the fields shown in Listing 3.19. Then to make this model used by the DSL use this model as the user model for the app, developers must implement the following configurations shown in Listing 3.20 in the app declaration. The entry `userModel` takes a reference to the user model as indicated by the name. While entries in lines 5-9 reference fields in the user model. The entry `onSuccessRedirectTo` is used to define where the user should be redirected to when they login to the app. While the entry `onFailRedirectTo` is used to define where the user should be redirected to when they try to access an app resource that requires authentication without being logged in.

```
model User {
  id         Int      @id
  username   String   @unique
  password   String
  isOnline   Boolean  @default(false)
  lastActive DateTime @default(Now)
  ...other fields...
}
```

Listing 3.19: User model example.

```
app AppWithAuth {
  title: "App with Authentication",
  auth: {
    userModel: User,
    idField: id,
    isOnlineField: isOnline,
    lastActiveField: lastActive,
    usernameField: username,
    passwordField: password,
    onSuccessRedirectTo: "/",
    onFailRedirectTo: "/login"
  }
}
```

Listing 3.20: Authentication configuration example.

To create sign up and login forms, developers can do that by creating components of type `SignupForm` or `LoginForm`. The `SignupForm` and `LoginForm` are created like `Create` components. The `SignupForm` must implement all the required fields of the user model. While `LoginForm` must implement the `username` and `password` fields only. The example in Listing 3.21 shows how a simple login form can be created. Moreover, `LogoutButton` are created like `Delete` components as shown in Listing 3.22.

```
component<LoginForm> MyLoginForm {
  formInputs: {
    username: {
      input: {
        type: TextInput,
        placeholder: "Enter username"
      }
    },
    password: {
      input: {
        type: PasswordInput,
        placeholder: "Enter password"
      }
    }
  },
  formButton: {
    name: "Login"
  }
}
```

Listing 3.21: Login form example.

```
component<LogoutButton> MyLogoutButton {
  formButton: {
    name: "Logout"
  }
}
```

Listing 3.22: Logout button example.

Users might need to grant some permissions before interacting with specific parts of the app. For example, users might need to be authenticated to view a certain page or do some operation. This can be achieved in the DSL by using the `@permissions` attribute and it can take either one of the following values `IsAuth` or `OwnRecord`. The `IsAuth` value makes sure that only authenticated users have permission to view this page, for example. While `OwnsRecord` is used to specify that users can do some operations only on their own records in the database. The permission attribute can be used only be used with `query` and `page` declarations. When it is used with the `query` declaration, it means that only users who have the specified permissions can do the operation. But, when it is used with the `page` declaration, it can only take `IsAuth` as a value and it means that only authenticated users can view the page.

The code in Listing 3.23 shows how the permissions attribute can be used with queries and pages by implementing a `FindMany` query which specifies that users can only fetch their own records from the `Post` model when they are authenticated. Furthermore, it implements a page that users can only view when they are authenticated.

```
@model(Post)
@permissions(IsAuth, OwnsRecord)
query<FindMany> getPostsWithPermissions { /** body */ }

@route("/posts")
@permissions(IsAuth)
page PostsListPage { /** body */ }
```

Listing 3.23: Permissions example.

3.7 Customization and Inline TypeScript

The DSL offers developers the option to write inline TypeScript code in the DSL. Also, the DSL allows developers to import JavaScript/TypeScript libraries. This allows developers to write custom code when the DSL becomes limiting. First, to import a JavaScript/TypeScript library developers have to specify the name of the package and the version in the `app` declaration as shown in Listing 3.24.

```
app AppWithTSCode {
  title: "App with TS code",
  serverDep: [
    ("is-even", "^1.0.0")
  ],
  clientDep: [
    ...client dependencies...
  ]
}
```

Listing 3.24: Importing a JavaScript library example.

The syntax for declaring custom queries and components in our DSL is the same as for default ones but they implement different entries. Both require a JSON-like object that must have a `fn` entry, which takes a TypeScript function as its value. Additionally, an optional entry can be included called `imports` to specify the imports required for the query or component. The TypeScript code must be defined inside the following notation `[| ... |]`.

The query custom function must be a TypeScript ExpressJS arrow function and it is an async function by default. Within the query function, developers can use Prisma using the Prisma client variable `prismaClient`. Prisma is the ORM used on the server to communicate with the database. Furthermore, the return value of all custom queries must be defined in the query declaration signature. If the return value of a query does not match a defined data model, developers can create custom types using a declaration called `type`. The `type` declaration provides a way to create custom types in the DSL to insure type safety of custom queries. All custom queries must have a route that specifies one of the following HTTP methods: get, post, put, and delete to use when calling this query. The custom query shown in Listing 3.25 receives a number from the client, checks if it is even, save the number in a table called `EvenNumber`, then returns `IsEvenMessage` back to the client.

Custom queries can be used with default components based on the HTTP method specified in the `@route` attribute. For instance, queries that implement the `post` method must be used in `Create` components. While queries that implement the `put` method must be used in `Update` components. Queries that implement the `get` method can be used with `FindUnique` and `FindMany` components based on their return value. If the return value is a list then the query can only be used with `FindMany` components, otherwise, it can only be used with `FindUnique` component. Finally, queries that implements the `delete` method must only be used with `Delete` components.

Custom components can take arguments, and be used in other components or pages, and any query defined in the app can be used in the custom components. Queries defined in the app can be accessed using the `Queries` object in the custom TypeScript code. The code in the `fn` entry should take a valid React functional component body code. For example, the code shown in Listing 3.26 implements a custom component that takes the username of the currently logged-in user as an argument and has a form that takes a number and submits it to the `isEven` query implemented in Listing 3.25. After submitting the form the component will show either “Even” or “Odd” to the user.

```
model EvenNumber {
  id     Int @id
  number Int
}

type IsEvenMessage {
  number  Int
  isEvent Boolean
}

@route("post", "/is-even")
query<Custom> isEven : IsEvenMessage {
  imports: [|
    import isEven from "is-even";
  |],
  fn: [|
    (req: Request, res: Response, next: NextFunction) => {
      // get the number from the request body
      const number = req.body.number;
      // check if the number is even using the is-even library
      if (isEven(number)) {
        // save the number in the EvenNumber table
        await prismaClient.evenNumber.create({
          data: { number }
        });
      }

      // send a response back to the client
      res.send({ number, isEven: isEven(number) });
    }
  |]
}
```

Listing 3.25: Custom query example.

```
component<Custom> IsEvenComponent(username: String) {
  imports: [|
    import React, { setState } from "react";
  |],
  fn: [|
    // declare form data local state
    const [fromData, setFormData] = setState<{ number: number }>({ number: null });
    //declare response data local state
    const [responseData, setResponseData] = setState<{
      number: number;
      isEven: boolean;
    }>({
      number: null,
      isEven: null
    });

    // updates the form data state with the change
    const handleInputChange = (event: React.ChangeEvent<HTMLInputElement>) => {
      setFormData({ number: event.target.number.value })
    }

    // submits the form data to the isEven query and updates the component state with the
        response
    const handleSubmit = async (event: React.FormEvent<HTMLFormElement>) => {
      // the query is used with the fetch API
      const response = await fetch(Queries.isEven(), {
        method: "post"
        body: JSON.stringify({ ...formData })
      });

      if (response.ok) {
        const data = await response.json();
        setResponseData(data);
      }
    }

    return (
      <>
        { responseData.isEven !== null ? (`Number ${responseData.number} is ${responseData.
            isEven ? "even" : "odd"}, ${username}`) : "" }
        <form onSubmit={handleSubmit}>
          <input
            type="number" name="number"
            value={responseData.number}
            onChange={handleInputChange}
          />
          <input type="submit" value="Submit"/>
        </form>
      </>
    )
  |]
}
```

Listing 3.26: Custom component example.

Chapter 4

Design and Implementation Details

In this chapter, we'll take a look on how we managed to compile to human-readable code and how the compiler's type checker helps preventing the impedance mismatch problem from happening. Additionally, we will explore the architecture of the compiled application in detail, examining how it is structured.

4.1 Compiler Architecture

The compiler architecture consists of three components: the frontend, AppSpecs generator, and code generator, as presented in Figure 4.1. Each of these components will be discussed in detail in the following sections. For further information on the source code of the compiler, please refer to Appendix 2.

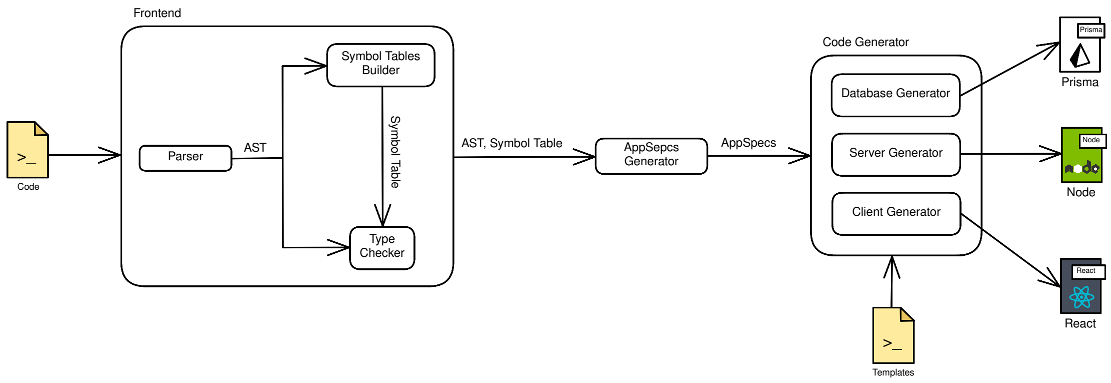


Figure 4.1: A diagram representing the compiler architecture.

Frontend

The frontend of the compiler is responsible for two main tasks: parsing the code and constructing an AST, as well as building a symbol table using the parsed AST. Once the symbol table has been built, the frontend conducts type-checking to ensure that the code is semantically correct. Since the DSL does not have a type inference feature, it requires type annotations to be included in the code. To prevent errors that could result in impedance mismatch within the application, the compiler performs a series of checks. These checks include but are not limited to:

- **UndefinedError**: This is a type of error that can occur when attempting to use an unbound declaration in general. For example, this error can occur when creating a `Create` component that specifies a form input for an undefined field in its `actionQuery`. Additionally, it can arise when the code is attempting to retrieve a non-existent field in a type that the query returns in the `FindMany` and `FindUnique` components. The purpose of this error is to alert developers that a particular tier of the application is trying to use an undefined resource from another tier or use an undefined declaration. For instance, the code shown in Listing 4.1 will raise this error because the `getExamples` query is trying to filter the `Example` model using an undefined field `title`.

```
model Example {
  id          Int     @id
  name        String
  anotherName String
}

@model(ExampleModel)
query<FindMany> getExamples {
  search: [ title ]
}
```

Listing 4.1: UndefinedError example.

- **QueryTypeError**: When a component receives a query of the wrong type, the compiler will raises this error. For instance, `Create` components always expect a query of type `Create` or `Custom` with an HTTP method of post. This error helps developers ensure that their code adheres to the expected query types for each component. For example, the code shown in Listing 4.2 will cause an error because the `ExamplesList` component is trying to fetch examples using a query of type `Create`.

```
@model(ExampleModel)
query<Create> createExample { /** body */ }

component<FindMany> ExamplesList {
  findQuery: createExample(),
  ...body...
}
```

Listing 4.2: QueryTypeError example.

- **RequiredFormInputError**: The compiler throws an error when the code of a `Create` or `Update` component does not specify an input for a required field in their `actionQuery`. For instance, if the `actionQuery` of a `Create` component requires two fields but the component only provides input for one of them, the compiler will raise an error. This error alerts developers to the missing input and helps ensure that their code correctly implements all required fields for the query. The code presented in Listing 4.3 will raise this error because the `ExampleCreateForm` does not implement inputs for all the fields required in the `createExample` query.

```
@model(ExampleModel)
query<Create> createExample {
  data: {
    fields: [ name, anotherName ]
  }
}

component<Create> ExampleCreateForm {
  actionQuery: createExample(),
  formInputs: {
    name: { /** body */ }
  }
}
```

Listing 4.3: RequiredFormInputError example.

- **FormInputTypeError**: This error occurs when the input of a form in a `Create` or `Update` component attempts to implement the wrong input type for a specific field. For example, if a data model has a field named `title` of type `String`, but the create form for the query that uses this data model implements an input of type `NumberInput`, the compiler will raise this error. The purpose of this error is to alert developers to the input type mismatch and ensure that the form inputs are correctly implemented to match the expected field types. The example in Listing 4.4 will arise this error because the form is trying to implement an input of type `NumberInput` for the field `name` of `ExampleModel` model implemented in Listing 4.1 which is of type `String`. Instead, this field should be of type `TextInput`.

```
component<Create> ExampleCreateForm {
  actionQuery: createExample(),
  formInputs: {
    name: {
      input: { type: NumberInput }
    }
  }
}
```

Listing 4.4: FormInputTypeError example.

By using these checks on the code, we prevent the impedance mismatch problem from occurring, and we also ensure that related runtime errors do not occur. For example, the app will not have an HTML form on the client that submits the wrong data to the server or a query on the server that tries to fetch or create a record from a data model using undefined fields or incorrect types. These checks help ensure that the app functions correctly without unexpected errors that can lead to poor user experience or application failures. There are many more error that the language's compiler can handle, please refer to Appendix 2 to learn more about them in the compiler's source code.

AppSpecs Generator

The AppSpecs generator uses the AST and symbol table to collect information about the database tables, backend API, and client app. Based on this information, it generates three different generic intermediate representations (IR), one for each part of the app. This IR contains common information extracted and derived from the code.

The IR for the backend API includes all the necessary information to create controllers, routes, and validators. Each endpoint in the API requires a controller function that handles the client request by executing a database query, a route that maps the request to the function, validation schemas, and middlewares to check permissions and validate data. The information contained in the IR is generic and can be used with any backend to generate code for nearly any framework.

For example, the IR for the query implemented in Listing 4.5, is represented in Figure 4.2. The AppSpecs generator uses the query definition to derive relevant data to construct the various parts of the IR. The route specifications are derived by determining the query's type, such as `FindUnique`, which indicates that the endpoint can be accessed using the GET HTTP method. Additionally, other information, such as the path, parameters, and required middleware, is derived from the query definition. Similarly, the controller function specifications are created to contain information about the inputs the query requires and other relevant details similar to the route specification. The validator specification includes information about the fields used in the query and their types. These details are later used to create validation schemas and generate type-safe backend APIs.

In addition, the AppSpecs generator creates IR for UI components and pages that can be used to compile the client and for data models to create database tables using the same technique. The generator identifies critical information for all parts of the application and stores them in data structures similar to the ones presented in Figure 4.2.

```
@model(ExampleModel)
@permissions(IsAuth, OwnsRecord)
query<FindUnique> getRecordById {
  where: id
}
```

Listing 4.5: FindUnique query.

query__id	getRecordById
path	example-model
params	(int, id)
http__method	GET
middlewares	[IsAuth, OwnsRecord]

(a) Route specifications.

query__id	getRecordById
query__type	FindUnique
model__name	exampleModel
where	true
search	false
data	false
owns__record	true

(b) Function specifications.

id	Int

(c) Validator specifications.

Figure 4.2: Endpoint IR example.

Code Generator

To generate the code, the code generator takes the IR generated in the previous step and applies it to the pre-written templates. The templates define the structure and logic of the generated code. The code generator is designed to be framework-agnostic, which means that it can generate code for any framework or library as long as the appropriate templates are available.

The code generator uses a template engine to merge the IR with the templates and produce the final code. The code in Listing 4.6, represents a simple template similar to the one implemented in the compiler using Jinja[9]. This template can be used to compile a NodeJS/ExpressJS controller function. In the example, we use the `query_id` as a name for the controller function. In the function, there is an object called `payload`, which is an object that stores the data sent in the HTTP request. It gets constructed based on the entries that the query implements. The pre-written templates include a middleware that is used to parse and validate the data in the HTTP request before executing the controller function, then it stores the result in the `req` object. We use the object `validatedPayload` to reconstruct a new payload object that can be used with the Prisma query. After that, we use the `prismaClient` to create a query based on the `model_name` and the `query_type`. Eventually, the function returns a response if the success or an error otherwise. Using this technique, we were able to compile human-readable code. This technique is used to compile all the parts in the app from the server to the client.

```
export const {{ query_id }} = async (req: Request, res: Response, next: NextFunction) => {
  try {
    const payload = {
      {% if where or search %}
      where: {
        {% if where %}
        ...req.validatedPayload?.where,
        {% endif %}

        {% if search %}
        ...req.validatedPayload?.search,
        {% endif %} },
      {% endif %}

      {% if data %}
        data: req.validatedPayload?.data,
      {% endif %}
    };

    const result =  await prismaClient.{{ model_name }}.{{ query_type }}(payload);

    res.status(httpStatus.OK).json(result);
  } catch (error) {
    next(error)
  }
}
```

Listing 4.6: Controller function template example.

4.2 Compiled App

As previously mentioned, the compiler compiles to a NodeJS server, a ReactJS client, and a Prisma Schema, which is used by Prisma ORM. In this section, we will delve into each component of the application, discussing its functionality and design decisions.

Database

The compiler does not compile directly to SQL but instead, it compiles to Prisma schema and Prisma queries. We are using Prisma ORM because it provides a way to create type-safe queries. The query builder API generates type-safe queries that are statically checked at compile-time, reducing the risk of runtime errors and making it easier to catch errors early in the development process. One of the main features that the DSL provides is compiling to human-readable code so that we can give developers the ability to take the compiled code to further develop it without using the DSL. That is why we are using Prisma because it let us make sure that the compiled app is well-typed. Also, Prisma provides an easy way to do database migration and data seeding. For example, when a table is updated with a new column, Prisma takes care of creating the new column and filling it with data if needed. The Prisma client is configured to use Postgresql database management but it can be edited easily to use any other relational database supported by Prisma.

NodeJS Server

The compiler creates a representational state transfer (REST) API, which is a popular architectural style for web development. REST APIs have been widely adopted by various industries and are used in real-world applications such as social media platforms. They enable access to resources over HTTP, using methods such as GET, POST, PUT, and DELETE to perform operations on resources, such as retrieving, creating, updating, and deleting them.

One of the key benefits of REST architecture is that the server is stateless, meaning it does not store information about previous client interactions or maintain an internal state. This helps in maintaining the reliability and availability of the server because it becomes easier to replicate and distribute the server across multiple machines. This, in turn, helps to prevent server crashes and outages. Additionally, statelessness enables easy scaling, as the server does not store any state for the clients. To scale the API, we can deploy it on multiple servers and implement a load-balancer to control traffic across them.

Due to the stateless nature of the server, authentication and authorization details are not stored on the server. Instead, the client sends authentication credentials with each request. We have implemented authentication using JSON Web Tokens (JWT), which generates a compact, self-contained token containing claims that identify the user and their permissions. After logging in or requesting access to a protected resource, the server returns the JWT to the client, which stores it in local storage for subsequent requests.

The DSL is designed to produce a type-safe REST API. This means that when a request is received from a client, the API verifies that the data is valid based on the specified data types and structures. To handle invalid data types or structures, the API includes various error-handling mechanisms that notify the client of the issue. For example, if the client sends incomplete data, the server returns an error message that specifies the nature of the issue, as shown in Listing 4.7. This error can also be triggered if the client sends data of an incorrect type, such as an integer value for a field that requires a value of type string. This feature is crucial because the server API can be accessed remotely, and requests can be sent without using the official client application. By including this type-safe feature, the API ensures that only valid data is received and processed.

```
// Returned when the client sends data with missing fields
{
  "status": 400,
  "code": "bad_request",
  "error": "invalid_input",
  "message": "Invalid inputs",
  "details": [
    {
      "code": "invalid_type",
      "parameter": "title",
      "message": "`title` is required"
    }
  ]
}

// Returned when the client sends data of the wrong type
{
  "status": 400,
  "code": "bad_request",
  "error": "invalid_input",
  "message": "Invalid inputs",
  "details": [
    {
      "code": "invalid_type",
      "parameter": "title",
      "message": "Expected string, received number"
    }
  ]
}
```

Listing 4.7: Errors example.

ReactJS Client

Finally, the compiler creates a ReactJS single page application (SPA), a type of web application that enhances the user experience by dynamically updating content on a single page without requiring multiple pages or complete page refreshes. Over the past decade, SPAs have gained popularity, especially with modern web development frameworks such as React, Angular, and Vue. Initially, when a user accesses an SPA, the server sends the HTML, CSS, and JavaScript files to the browser, and then the JavaScript code takes over and interacts with the server through the backend API to retrieve or modify data without a full-page reload. The primary advantage of SPAs is their ability to reduce the server load by only sending the initial files, resulting in faster load times and more responsive user interfaces. Additionally, SPAs offer a native application-like feel, providing more interactive and engaging user interfaces. Similar to the server, the client app includes a validation layer on HTML forms, alerting users of any incorrect data entry before making a request to the server.

Real-time Apps by Default

Real-time web applications require a constant flow of data between the client and the server. There are different techniques that can be used to achieve this feature. SSE allows the server to send real-time updates to the client using a single, long-lived connection. WebSockets provide a bidirectional communication channel between the client and server, allowing for real-time data exchange. Long polling involves the client making a request to the server and waiting for a response, keeping the connection open until new data is available. Short polling, on the other hand, involves the client regularly polling the server for new data.

Polling is not as practical as SSE and WebSockets because it can result in a high volume of unnecessary network traffic and latency issues, especially when there are many clients. With polling, the client sends requests to the server at regular intervals, regardless of whether new data is available. This can cause a large amount of data to be transmitted, even if there is no new information to send, which can result in increased network traffic and slower response times.

In contrast, SSE and WebSockets use a push-based approach where the server sends updates to the client only when new data is available. This eliminates the need for the client to repeatedly request new data, reducing the amount of network traffic and improving response times. Additionally, SSE and WebSockets both allow for real-time updates, making them more suitable for applications that require real-time data exchange.

After careful consideration, we opted to use SSE over WebSockets for the following reasons:

- **Simplicity**: SSE is simpler to implement than WebSockets, making it a good choice for applications that do not require bidirectional communication. SSE uses a simple HTTP connection to send data from the server to the client, while WebSockets require a dedicated WebSocket connection to enable bidirectional communication.
- **Automatic reconnection**: SSE supports automatic reconnection, which means that if the connection is lost, the client will automatically reconnect to the server without any user intervention. This makes SSE more reliable than WebSockets in some cases, as WebSockets may require additional code to handle reconnection.
- **Lightweight**: SSE is a lightweight protocol that requires less overhead than WebSockets, making it a good choice for applications that need to conserve bandwidth or have limited server resources.

To make the compiled app real-time using SSE, the client sends an initial request to the server when a page requiring data is mounted. This request loads all the necessary data and then caches it locally. After that, the client establishes a persistent connection with the server to listen for any updates to this data. Whenever there is a mutation on the server, the server sends an event to all connected clients containing the updated data. Clients then update their local state using this event, ensuring that their data is up-to-date with the server. This allows for real-time updates to be pushed to clients without the need for the client to repeatedly poll the server for new data.

4.3 Syntax Highlighting

The DSL has a Visual Studio Code (VSCode) extension that provides syntax highlighting. This feature is crucial for a positive developer experience because it makes the language's syntax more readable and understandable. Syntax highlighting provides visual cues for different parts of the code, such as keywords, variables, and comments. It also makes it easier to spot syntax errors and other issues. The extension is implemented using TextMate grammar, which is the standard format that VSCode uses for syntax highlighting. TextMate grammar is widely used by other famous text editors like Sublime Text and Atom. This makes it easy for us to support syntax highlighting in these editors too.

Chapter 5

Evaluation

We will demonstrate the DSL's effectiveness in building fully-functional web apps with varying requirements and features. We will then conduct user experiments to evaluate the learning curve, syntax, and developer experience of the DSL among developers with varying levels of experience. The data collected will include the time taken to complete each task, ease of understanding the DSL syntax, and any challenges encountered. Finally, we aim to identify and address any limitations or issues of the DSL to inform future development and improvements.

5.1 Evaluation through Application Development

The DSL aims to streamline the web app development process, minimize errors, and reduce the time and effort required. To showcase the language's effectiveness, we have developed 3 web apps that exemplify its capabilities. Each app highlights distinct features of the DSL, illustrating how it can rapidly produce functional web apps. The source code for all the 3 applications mentioned below can be found in Appendix 3.

Todo App

The Todo app is designed for creating, updating, and deleting tasks while incorporating an authentication feature to restrict unauthorized access and actions. Its interactive interface provides real-time updates, eliminating the need for page refreshes. One of the significant benefits of using the DSL to build the Todo app is its capability to implement the permissions feature easily. By using the `@permissions` attribute, it was effortless to specify that only authorized users could access their tasks while restricting access to tasks created by other users. As shown in Figure 5.1, there are two users using the app and each of them can only see their own tasks. This streamlined approach simplifies the development process and ensures secure and efficient functionality.

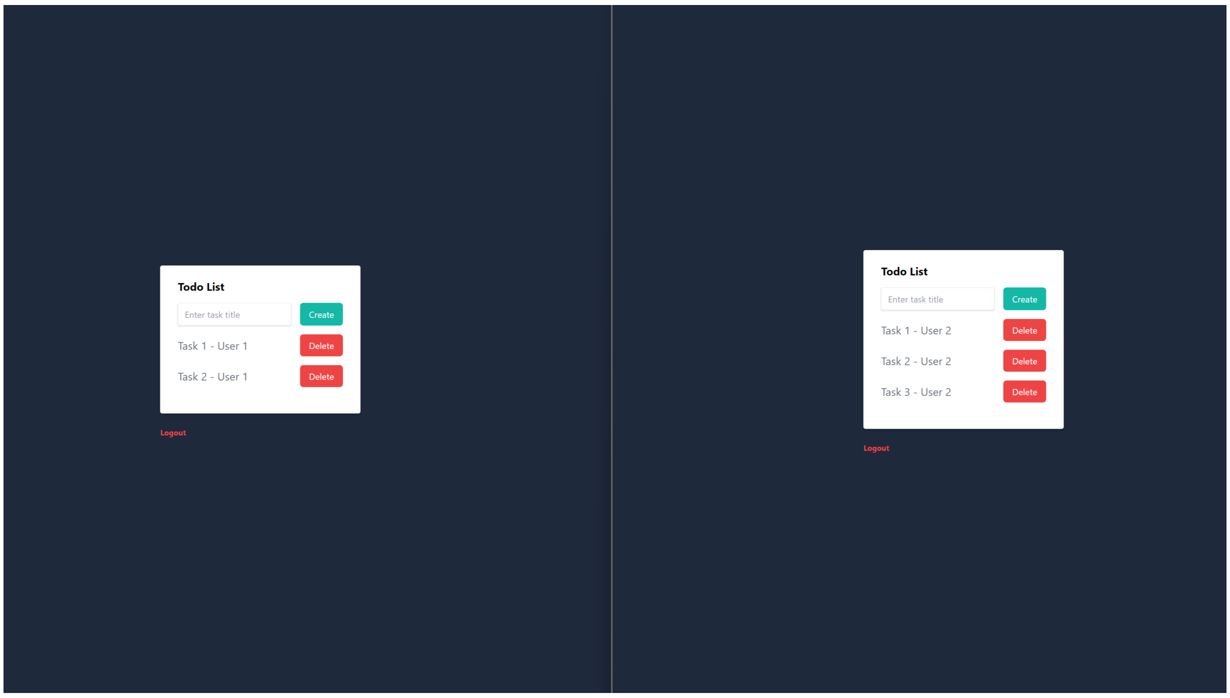
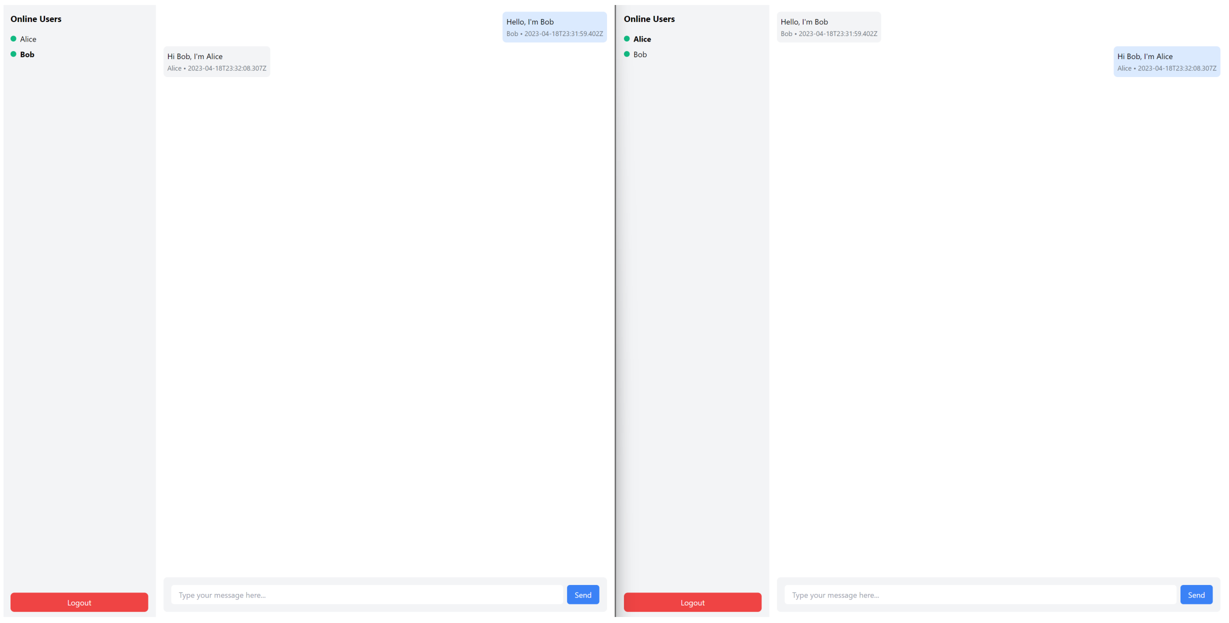


Figure 5.1: Todo app user interface.

Chatroom

The Chatroom app provides users with the ability to create accounts and engage in conversations with other users. It also includes a user list that displays all active users along with their online/offline status. The app leverages real-time updates, eliminating the need for manual refreshing. These updates are facilitated by a server that pushes events to clients using SSE, with the clients automatically updating their local state in real-time. The UI for this application is shown in Figure 5.2, where we have two users Alice and Bob messaging each other, and the state of the client app is updated in real-time. Whenever any of them sends a message the message will be loaded for the other user automatically without refreshing the page, as well as the status of each user will be updated. This app can have many users chatting in the same chatroom and the app will update the state of all the clients in real-time.

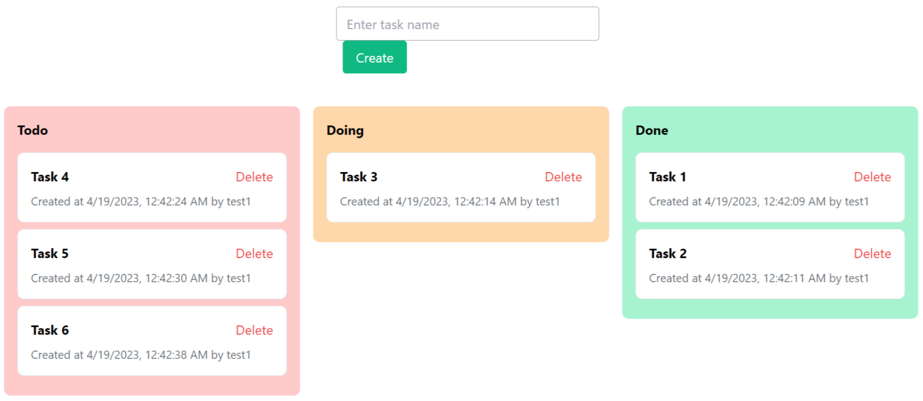

Hello, I'm Bob
Bob • 2023-04-18T23:31:59.402Z

Hi Bob, I'm Alice
Alice • 2023-04-18T23:32:08.307Z

Figure 5.2: Chatroom user interface.

Kanban Board

The Kanban board app is a collaborative app that allows different users to use a Kanban board to organize their tasks together. The app implements three columns: To do, In progress, and Done. Users can create new tasks and they can drag and drop any task from one column to another to update its state. Like the previously mentioned apps, all updates in this app happen in real-time. Moreover, this app shows that developers can interop with TypeScript and use libraries available in the JavaScript/TypeScript ecosystem. To implement this app, we used a library called react-beautiful-dnd[10], which allowed us to create the drag-and-drop feature in the app easily. As shown in Figure 5.3 users can create tasks and drag them between the different columns. Like the chatroom app, the state of this app is updated between all the clients logged in automatically in real-time.



Todo

Task 4 — Created at 4/19/2023, 12:42:24 AM by test1

Task 5 — Created at 4/19/2023, 12:42:30 AM by test1

Task 6 — Created at 4/19/2023, 12:42:38 AM by test1

Doing

Task 3 — Created at 4/19/2023, 12:42:14 AM by test1

Done

Task 1 — Created at 4/19/2023, 12:42:09 AM by test1

Task 2 — Created at 4/19/2023, 12:42:11 AM by test1

Figure 5.3: Kanban board user interface.

5.2 Evaluation on Requirements

The DSL simplifies web development by abstracting many low-level details, allowing developers to focus on high-level functionality and features. We evaluated the effectiveness of the DSL by comparing the lines of code required to build each app using the DSL versus using pure NodeJS and ReactJS. Table 5.1 shows that developers can create the Todo app and Chatroom app with just 15% of the lines of code required for NodeJS and ReactJS. While apps that require TypeScript, like the Kanban Board app, require slightly more lines of code in the DSL, the total is still much less than the lines of code required for pure NodeJS/ReactJS development.

Table 5.1: Lines of code required to implement each app in the DSL and NodeJS/ReactJS including empty lines and excluding build configuration files.

App	DSL	NodeJS/ReactJS
Todo App	233	1547
Chatroom	258	1580
Kanban Board	405	1669

The DSL has been shown to meet all the requirements outlined in Chapter 2 through the development of the three discussed apps.

- **R1. Support multitier programming**: We were able to build all three apps without the need for tiers because the compiler was responsible for tier splitting by leveraging the different declarations in the DSL. This allowed us to solve the impedance mismatch problem that is commonly encountered in web development. Additionally, the DSL's static-typing feature made it possible for the compiler to catch errors at compile time, reducing the risk of encountering runtime errors.
- **R2. Allow developing full-stack apps declaratively**: The use of declarations allowed us to abstract away low-level details, providing developers with high-level declarations to use instead. As a result, building all three apps in the DSL was significantly easier compared to using pure NodeJS and ReactJS, as shown in Table 5.1. Also, using the different declarations in the DSL, the compiler was able to split the code into different tiers at compile time.
- **R3. Compile to human-readable and type-safe TypeScript code using existing tech stacks**: Using the generated IR and utilizing the pre-written templates, we managed to compile to a NodeJS server and a ReactJS client, producing human-readable and type-safe TypeScript code. As we discussed in Chapter 4.
- **R4. Real-time by default**: The Chatroom and Kanban Board apps both feature real-time updates, with the client state syncing between all connected clients. This was made possible through the use of SSE, which is explained in detail in Chapter 4.
- **R5. Allow interoperability with TypeScript**: The DSL provides developers with the ability to write inline TypeScript code using `Custom` queries and components. This feature allows for greater flexibility in application development and customization. Moreover, specifying JavaScript/TypeScript packages in the app configuration enables easy interoperability with TypeScript, as demonstrated in the Kanban Board app.

5.3 User Evaluation

The online experiment conducted to evaluate the learning curve, syntax, and developer experience of the DSL was divided into two parts, each assessing different aspects of the participants' experience. In the first part, participants were presented with a working code snippet without any prior instruction on the DSL, aimed at evaluating their understanding of the code and assessing its comprehensibility. Each question in this section was divided into two parts: First, participants were asked to rate their understanding of the code snippet on a scale from 1 to 5, where a higher number indicated a better understanding of the code snippet. Second, they were asked to explain what they understood from the code snippet. This approach effectively measured the initial learning curve and syntax of the language.

The second part aimed to assess the developer experience by providing participants with a code snippet containing a bug and the corresponding compiler error message. Participants were then required to explain the error in the code and suggest a solution. The experiment's design ensured a diverse population of participants by asking about their programming and web development experience before the experiment.

After the experiment was completed, participants were allowed to reflect on their overall experience with the DSL and provide feedback on the language aspects that they found most appealing or confusing. This approach allowed us to gain valuable insights into the effectiveness of the DSL and identify areas that required improvement.

Of the 9 participants who took part in the experiment, most were experienced programmers, although not all had experience with web development. Impressively, 5 out of the 9 participants were able to explain all of the presented code snippets correctly, while the remaining 4 participants correctly answered an average of 8.75 questions. These results suggest that the language syntax is easy to comprehend and that the compiler messages were effective in identifying errors.

It is notable that, on average, it took the participants 33 minutes to solve all the questions, indicating that they were able to grasp the language's syntax quickly simply by reading it without any prior exposure to DSL. This highlights the language's intuitive nature and demonstrates the participants' ability to quickly comprehend the language's syntax.

The use of the `@` annotation to declare attributes were found to be confusing by many participants. Similarly, the `<>` notation used to specify the types of queries and UI components was found to be confusing by several participants, as it is commonly used to define type parameters in many programming languages.

A few participants had difficulty understanding the process of defining one-to-many relations, particularly in determining which parameter passed to the `@relation` attribute represents the foreign key. The `connect` notation used to connect records also proved challenging for some participants.

Despite encountering some confusing notations and not having received any training or documentation on the DSL, the majority of the participants were able to answer all questions correctly. This highlights the DSL's accessibility and user-friendly nature. Almost all of the participants found the syntax of the language easy to interpret, while the compiler messages were found to help identify and correct errors in the code. For more information about the questions asked and the data collected from the experiment, please refer to Appendix 4.

5.4 Issues

In this section, we will explain some issues related to the DSL. These issues include those related to the architecture of the compiler as well as those related to the application that the language compiles.

Problems with REST APIs

REST APIs present several challenges that can impact their performance and reliability. One common issue is returning excessive data that the client does not need, leading to large data responses and increased latency. Additionally, REST APIs may struggle to handle high levels of traffic, which can result in slow response times and even system crashes. To address this issue, it may be necessary to increase the number of servers running the API and implement efficient load-balancing and caching strategies to ensure optimal performance and reliability.

Problems with SPAs

One of the most significant issues with SPAs is their impact on search engine optimization (SEO). Search engines like Google rely heavily on content and links to index and rank websites. However, since SPAs only load a single HTML file and dynamically update the content using JavaScript, search engine crawlers have a difficult time indexing the content of the page[11]. This can result in a lower search engine ranking, making it harder for users to find the website.

Although it is a significant problem in SPAs, there are techniques to solve it, such as server-side rendering (SSR) and static site generation (SSG). In SSR, the server processes and renders the webpage before sending it to the client. When a user requests a page, the server generates the HTML, CSS, and JavaScript required for the page and sends it to the client. This approach means that the browser receives a fully rendered page that is ready to be displayed, reducing the amount of work the browser needs to do and improving the page load time. On the other hand, SSG generates the HTML, CSS, and JavaScript for a website at build time, rather than at runtime. The resulting files are then hosted on a web server and served to users as static pages. This approach allows for faster load times since the pages are pre-rendered and do not require any server processing when a user requests them. Additionally, it improves SEO, since search engines can easily crawl and index static pages. Therefore, sites that use SSR or SSG have the potential to rank higher in search engine results.

Cannot Compile to All Existing Framework

Generally, the AppSpecs are designed to be framework-agnostic, but some parts of it are generated specifically to work with certain frameworks and libraries. For instance, UI structures are translated to JSX[12], an XML-like language used by various UI frameworks such as ReactJS, Preact, SolidJS, and Qwik. However, not all UI frameworks use JSX; Angular and Svelte, for instance, use their own template syntax. As a result, the compiler's IR can only be utilized to generate code for frameworks that support JSX.

Unsafe TypeScript Interoperability

The DSL cannot currently check the inline TypeScript code and instead treats it as a string. This can lead to various issues, especially if the data returned by the inline code does not match the data specified in the query signature. Such inconsistencies can result in runtime errors and cause significant problems.

Therefore, it is essential to exercise caution when working with inline TypeScript code in the DSL and ensure that the returned data aligns correctly with the expected data type.

Chapter 6

Conclusion

In conclusion, we successfully created a DSL that generates human-readable code for a NodeJS server and a ReactJS client. With the use of declarations, the compiler can easily identify the code for each tier in the app. The DSL also abstracts many of the low-level implementation details required in modern web apps, resulting in faster development without the need for boilerplate code. We demonstrated this by building simple Todo and Chatroom apps that required only 15% of the lines of code needed in NodeJS and ReactJS. The compiler's type-checking feature identifies impedance mismatch related problems at compile time, leading to more efficient web apps. Using the techniques explained for generating IR and app code, we managed to make the compiler compile to TypeScript human-readable code. Moreover, by leveraging template engines, we managed to make our compiler framework agnostic. The DSL generates real-time apps by default using SSE. Also, we have shown that the language can interop with TypeScript and make use of existing tools in the JavaScript/TypeScript ecosystem by developing the Kanban app. This makes it more adoptable than previous multitier programming languages. The declarative approach implemented in the language has made its code more readable and easier to debug compared to other multitier languages that use annotations. Additionally, we have implemented tools such as expressive error messages in the compiler and syntax highlighting to enhance the developer experience. Finally, we demonstrated in our user experiment that the DSL has a low learning curve.

6.1 Future Work

The DSL allows for the creation of simple queries, but it cannot implement more complex queries. For instance, some apps may require a query that paginates the data list returned by the `FindMany` query. However, the DSL cannot paginate data by limiting the number of records that a query can return. Additionally, bulk operations are not supported, meaning that the DSL lacks the feature to create, update, or delete multiple records simultaneously.

As previously mentioned, the DSL's current compilation to an SPA client presents limitations in terms of SEO. To overcome this obstacle, it would be advantageous for the DSL to incorporate SSR/SSG mechanisms for UI components. This addition would allow search engines to index and crawl the website's content, boosting its visibility and search ranking.

From the user experiment conducted, we found that some of the notations used in the DSL were confusing to some participants. While some notations, such as the `@` and `connect` notation, could have been clearer to participants if they had received documentation on the DSL before the experiment, others, like the `<>` notation, which is used in many languages to specify type parameters, were particularly confusing. In addition, defining one-to-many relations proved difficult for some participants. To improve the DSL's syntax, we should consider creating a new notation to define query and UI component types. Additionally, we could make the `@relation` attribute take named parameters to make it clearer.

Chapter 7

Appendix 1: Language Grammar

```
capitalized_identifier ::= [A-Z_][A-Za-z0-9_]*

identifier ::= [A-Za-z_][A-Za-z0-9_]*

string_literal ::= ''' [^']* '''

boolean_literal ::= true | false

integer_literal ::= digit+

digit ::= ['0'-'9']

variable_literal ::= identifier ('.' identifier)*

literal = string_literal | boolean_literal | integer_literal | variable_literal

type ::= 'String' | 'Int' | 'Boolean' | 'DateTime' | capitalized_identifier

parameter ::= identifier ':' type

connect_with_expr ::= 'connect' variable_literal 'with' variable_literal

permission ::= 'IsAuth' | 'OwnsRecord'

permissions_attribute ::= '\@permissions' '(' permission (',' permission)* ')'

http_method ::= 'get' | 'post' | 'put' | 'delete'

route_attribute ::= '\@route' '(' (http_method ',')? string_literal ')'

model_attribute ::= '\@model' '(' capitalized_identifier ')'

custom_entries ::= 'fn' ':' '[|' string_literal '|]' (',' 'imports' ':' '[|' string_literal '|]
    ')?
```

Listing 7.1: General rules

App Declaration Grammar

```
app_declaration ::= 'app' capitalized_identifier '{' app_entries '}'

app_entries ::= 'title' ':' string_literal (',' auth_entry)?

auth_entry ::= 'auth' ':' '{' auth_entries '}'

auth_entries ::= 'userModel' ':' capitalized_identifier ','
                 'idField' ':' identifier ','
                 'isOnlineField' ':' identifier ','
                 'lastActiveField' ':' identifier ','
                 'usernameField' ':' identifier ','
                 'passwordField' ':' identifier ','
                 'onSuccessRedirectTo' ':' string_literal ','
```

```
                'onFailRedirectTo' ':' string_literal
```
Listing 7.2: App declaration rules

Data Models Grammar

```
model_name ::= 'model' capitalized_identifier '{' model_fields '}'

model_fields ::= model_field (',' model_field)*

model_field ::= identifier type field_attributes*

field_attributes ::= '\@id' | '\@unique' | '\@default(value)' | '\@updatedAt'
```
Listing 7.3: Model declaration rules

Queries Grammar

```
query_declaration ::= query_attributes+ 'query' '<' query_type '>' identifier '{' query_entries
     '}'

query_attributes ::= permissions_attribute | model_attribute | route_attribute

query_entries = where_entry | search_entry | data_entry | custom_entries

query_type ::= 'Create' | 'FindUnique' | 'FindMany' | 'Update' | 'Delete' | 'Custom'

where_entry ::= identifier

search_entry ::= '[' identifier+ ']'

data_entry ::= 'fields' ':' '[' identifier+ ']' (',' relation_field_entry)?

relation_field_entry ::= 'relationFields' ':' '{' relation_field (',' relation_field)* '}'

relation_field ::= identifier ':' connect_with_expr
```
Listing 7.4: Query declaration rules

XRA Grammar

```
html_tag ::= SET_OF_HTML_TAGS

xra_element_name ::= html_tag | capitalized_identifier

xra_element ::= self_closing_xra_element | xra_element_opening xra_children xra_element_closing
     | xra_for_expression | xra_if_expression

xra_fragment ::= '<>' xra_element+ '</>'

xra_children ::= xra_element | xra_literal_expression

xra_element_opening ::= '<' xra_element_name xra_element_attribute* '>'

xra_element_closing ::= '</' xra_element_name '>'

self_closing_xra_element ::= '<' xra_element_name xra_element_attribute* '/>'

xra_element_attribute ::= identifier '=' xra_literal_expression | string_literal |
    boolean_literal | integer_literal

xra_literal_expression ::= '{' literal '}'

xra_for_expression ::=
  '[%' 'for' identifier 'in' identifier '%]'
    xra_element | xra_fragment
  '[% endfor %]'

xra_if_expression ::=
  '[%' 'if' condition '%]'
    xra_element | xra_fragment
  '[%' 'endif' '%]'

```

```
condition ::= literal logical_operation literal

logical_operation ::= '==' | '<' | '>' | '<='  | '>='

render_expression ::= xra_element+ | xra_fragment+
```

Listing 7.5: XRA rules

Component Grammar

```
component_declaration ::= 'component' '<' component_type '>' capitalized_identifier ('('
    parameter* ')')? { component_body }

component_body ::= render_expression | fetch_component_entries | action_form_component_entries
    | action_button_component_entries | custom_entries

component_type ::= 'Create' | 'FindUnique' | 'FindMany' | 'Update' | 'Delete' | 'Custom' | '
    SignupForm' | 'LoginForm' | 'LogoutButton'

find_component_entries ::=
  'findQuery' ':' find_query ','
  'onError' ':' render_expression ','
  'onLoading' ':' render_expression ','
  'onSuccess' ':' render_expression

find_query ::= identifier '(' ')'

find_query_entries ::= '{'
  'where' ':' '{' where_or_search_field+ '}'
  | 'search' ':' '{' where_or_search_field+ '}'
'}'

where_or_search_field ::=  identifier ':' literal

action_form_component_entries ::=
  'actionQuery' ':' identifier '()' ','
  'formInputs' ':' '{' form_input_entry '}' ','
  'formButton' ':' '{' form_button_entry '}'
  (',' 'globalStyle' ':' '{' global_style_entry '}')?

action_button_component_entries ::=
  'actionQuery' ':' identifier '()' ','
  form_name_entry
  (',' form_style_entry)?

form_input_type ::= 'TextInput' | 'EmailInput' | 'PasswordInput' | 'NumberInput' | 'NumberInput
    ' | 'CheckboxInput' | 'DateTimeInput' | 'DateInput'

global_style_entry ::= 'formContainer' ':' string_literal
                     | 'inputContainer' ':' string_literal
                     | 'input' ':' string_literal
                     | 'inputError' ':' string_literal
                     |  'inputLabel' ':' string_literal

form_input_entry ::= identifier ':' '{' form_input_entry_options '}'

form_input_entry_options ::=
'input' ':'
  '{' form_input_type_entry
      (',' form_input_default_value
      ',' form_input_placeholder
      ',' form_input_visibility
      ',' form_style_entry)? '}'
    (',' 'label' ':'
      '{' form_name_entry (',' form_style_entry )? '}')?
    (',' form_style_entry)?

form_input_type_entry ::= 'type' ':' form_input_type

form_input_default_value ::= 'defaultValue' ':' literal | connect_with

form_input_visibility ::= 'isVisible' ':' boolean_literal

```

```
form_input_placeholder ::= 'placeholder' ':' string_literal

form_button_entry ::= '{' form_name_entry (',' form_style_entry)? '}'

form_name_entry ::= 'name' ':' string_literal

form_style_entry ::= 'style' ':' string_literal
```

Listing 7.6: Component declaration rules

Page Declaration

```
page_declaration ::= page_attributes+ 'page' capitalized_identifier ('(' parameter* ')')? {
    render_expression }

page_attributes ::= permissions_attribute | route_attribute
```

Listing 7.7: Page declaration rules

Chapter 8

Appendix 2: Source Code

Prerequisites

Before you can build and run this code, you will need to have the following installed on your system:

1- OCaml = v4

2- Dune = v3.6

3- NodeJS = v16

4- PostgreSQL = v14.7

Building

To build the compiler, please follow the instructions below:

```
dune build bin/main.exe
```

Running To run the compile, please follow the instructions below:

Any app can be compiled using the following command.

```
dune exec -- bin/main.exe PATH_TO_APP DATABASE_NAME
```

Command Line Options

The following command line options are available for this project:

- –output_dir: The path to the directory that will contain the compiled code. Default value: ".out"
- –server_port: The port used to run the server on. Default value = 4000

To use these options, simply include them when compiling the app from the command line, as follows:

```
dune exec -- bin/main.exe PATH_TO_APP DATABASE_NAME --output_dir=PATH_TO_DIR --server_port=
    PORT_NUMBER
```

The compiler creates two different directories, one for the server and another for the client. To run the compiled app, first, ensure that the PostgreSQL server is running on the machine. Next, navigate to the server directory and use the following command to create the database:

```
yarn prisma migrate dev
```

After that, you can run the server using the following command:

```
yarn dev
```

To run the client, navigate to the client directory and use the same command that was used to run the server.

Folder Structure

```
/main_dir
  /.vscode // contains settings for vscode
  /bin // contains the executable file for the compiler
  /examples // contains examples for apps built using the DSL
  /language_tools // contains the code for the VSCode syntax highlighting extension
  /src // contains the main code for the compiler
  |  /analyzer // contains the code for the lexer, parser, and the type checker
  |  |  /ast // the package responsible for maintaining and formatting the AST
  |  |  /error_handler // the package responsible for creating error messages
  |  |  /parsing // the package responsible for lexing and parsing the code
  |  |  /type_checker // the package used for type checking
  |  |  |  checker.ml // the main checker file, it calls the symbol table builder and the
      checker
  |  |  |  environment.ml // the file that defines the symbol table data structure
  |  |  |  model_check.ml // implements the code responsible for checking data models
  |  |  |  query_checker.ml // implements the code responsible for checking queries models
  |  |  |  type_decl_checker.ml // implements the code responsible for checking the type
      declaration
  |  |  |  xra_checker.ml // implements the code responsible for checking the UI declarations
  |  /generator // the package used to generate the code
  |  /specs // the package used to generate the IR
  /templates // contains the templates for the db, client, server
  |  /client
  |  /db
  |  /server
```

Chapter 9

Appendix 3: Applications Source Code

Todo App

```
app Todo {
    title: "Todo App Created by Ra",
    auth: {
        userModel: User,
        idField: id,
        isOnlineField: isOnline,
        lastActiveField: lastActive,
        usernameField: username,
        passwordField: password,
        onSuccessRedirectTo: "/",
        onFailRedirectTo: "/login"
    }
}

model User {
    id          Int        @id
    username    String     @unique
    password    String
    isOnline      Boolean      @default(false)
    lastActive DateTime  @default(Now)
    tasks         Task[]
}

model Task {
    id         Int       @id
    title      String
    isDone     Boolean   @default(false)
    user             User         @relation(userId, id)
    userId       Int
    createdAt DateTime @default(Now)
    updatedAt DateTime @updatedAt
}

@model(Task)
@permissions(IsAuth, OwnsRecord)
query<FindMany> getTasks {
    search: [ title, isDone ]
}

@model(Task)
@permissions(IsAuth)
query<Create> createTask {
    data: {
        fields: [title],
        relationFields: {
            user: connect id with userId
        }
    }
}

@model(Task)
@permissions(IsAuth, OwnsRecord)
query<Delete> deleteTaskById {
    where: id
}

component<Create> TaskCreateForm {
    actionQuery: createTask(),
    globalStyle: {
        formContainer: "flex mt-4"
    },
    formInputs: {
        title: {
            style: "flex w-full",
            input: {
                type: TextInput,
                placeholder: "Enter task title",
                isVisible: true,
                style: "shadow border rounded py-2 px-3 w-full mr-4 text-grey-darker"
            }
        },
        user: {
            input: {
                type: RelationInput,
                isVisible: false,
                defaultValue: connect id with LoggedInUser.id
            }
        }
    },
    formButton: {
        name: "Create",
        style: "rounded-md bg-teal-500 text-white px-4 py-2"
    }
}

component<Delete> TaskDeleteButton(id: Int) {
    actionQuery: deleteTaskById({
        where: id
    }),
    formButton: {
        name: "Delete",
        style: "rounded-md bg-red-500 text-white px-4 py-2"
    }
}

component TaskComponent(task: Task) {
    render(
        <div className="flex mb-4 items-center">
            <div className="w-full text-gray-500 text-xl">{ task.title }</div>
            <TaskDeleteButton id={task.id} />
        </div>
    )
}

component<FindMany> TasksComponent {
    findQuery: getTasks() as tasks,
    onError: render(<div>{ "An error occurred" }</div>),
    onLoading: render(<div>{ "Loading..." }</div>),
    onSuccess: render(
        <>
            [% for task in tasks %]
                <TaskComponent task={ task } />
            [% endfor %]
        </>
    )
}

component<SignupForm> MySignupForm {
    globalStyle: {
        formContainer: "bg-white shadow-md rounded px-8 pt-6 pb-8 mb-4",
        input: "appearance-none border rounded w-full py-2 px-3 text-gray-700 leading-tight
            focus:outline-none focus:shadow-outline",
        inputLabel: "block text-gray-700 font-bold mb-2"
    },
    formInputs: {
        username: {
            label: {
                name: "Username"
            },
            input: {
                type: TextInput,
                placeholder: "Enter username"
            }
        },
        password: {
            label: {
                name: "Password"
            },
            input: {
                type: PasswordInput,
                placeholder: "Enter password"
            }
        }
    },
    formButton: {
        name: "Signup",
        style: "bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded focus:
            outline-none focus:shadow-outline"
    }
}

component<LoginForm> MyLoginForm {
    globalStyle: {
        formContainer: "bg-white shadow-md rounded px-8 pt-6 pb-8 mb-4",
        input: "appearance-none border rounded w-full py-2 px-3 text-gray-700 leading-tight
            focus:outline-none focus:shadow-outline",
        inputLabel: "block text-gray-700 font-bold mb-2"
    },
    formInputs: {
        username: {
            label: {
                name: "Username"
            },
            input: {
                type: TextInput,
                placeholder: "Enter username"
            }
        },
        password: {
            label: {
                name: "Password"
            },
            input: {
                type: PasswordInput,
                placeholder: "Enter password"
            }
        }
    },
    formButton: {
        name: "Login",
        style: "w-full bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded
            focus:outline-none focus:shadow-outline"
    }
}

component<LogoutButton> MyLogoutButton {
    formButton: {
        name: "Logout",
        style: "inline-block align-baseline font-bold text-sm text-red-500 hover:text-red-800"
    }
}

@route("/")
```

```
@permissions(IsAuth)
page Tasks {
    render(
        <div className="flex h-screen items-center justify-center bg-slate-800">
            <div className="space-y-2">
                <div className="bg-white shadow-md rounded px-8 pt-6 pb-8 mb-4">
                    <div className="mb-4">
                        <h1 className="text-grey-darkest text-xl font-bold mb-4">{"Todo List"
                            }</h1>
                        <TaskCreateForm />
                    </div>
                    <TasksComponent />
                </div>
                <MyLogoutButton />
            </div>
        </div>
    )
}

@route("/login")
page LoginPage {
    render(
        <div className="flex h-screen items-center justify-center bg-slate-800">
            <div className="space-y-2">
                <span className="text-xl text-white">{"Login form"}</span>
                <MyLoginForm />
                <a className="inline-block align-baseline font-bold text-sm text-blue-500 hover
                    :text-blue-800" href="/signup">
            { "Sign Up" }
            </a>
            </div>
        </div>
    )
}

@route("/signup")
page SignupPage {
    render(
        <div className="flex h-screen items-center justify-center bg-slate-800">
            <div className="space-y-2">
                <span className="text-xl text-white">{"Sign up form"}</span>
                <MySignupForm />
                <a className="inline-block align-baseline font-bold text-sm text-blue-500 hover
                    :text-blue-800" href="/signup">
            { "Login" }
            </a>
            </div>
        </div>
    )
}
```

Listing 9.1: Todo app source code.

Chatroom

```
app ChatRoom {
    title: "Chat Room Created by Ra",
    auth: {
        userModel: User,
        idField: id,
        isOnlineField: isOnline,
        lastActiveField: lastActive,
        usernameField: username,
        passwordField: password,
        onSuccessRedirectTo: "/",
        onFailRedirectTo: "/login"
    }
}

model User {
    id          Int        @id
    username    String     @unique
    password    String
    isOnline     Boolean     @default(false)
    lastActive DateTime  @default(Now)
    messages   Message[]
}

model Message {
    id        Int      @id
  content   String
    user            User        @relation(userId, id)
    userId      Int
    createdAt DateTime @default(Now)
    updatedAt DateTime @updatedAt
}

@model(Message)
@permissions(IsAuth)
query<Create> createMessage {
    data: {
        fields: [content],
        relationFields: {
            user: connect id with userId
        }
    }
}

@model(Message)
@permissions(IsAuth)
query<FindMany> getMessages {
    search: [content]
}

@model(User)
@permissions(IsAuth)
query<FindMany> getUsers {
    search: [username]
}

component<FindMany> MessagesComponent {
    findQuery: getMessages() as messages,
    onError: render(
        <div>{ "An error occured" }</div>
    ),
    onLoading: render(
        <div>{ "Loading..." }</div>
    ),
    onSuccess: render(
    <div className="flex flex-col gap-2 flex-1">
            [% for message in messages %]
                <div
                    key={message.id}
                    className=[% if message.userId == LoggedInUser.id %]
                                            { "flex gap-2 justify-end" }
                                        [% else %]
                                            { "flex gap-2" }
                                        [% endif %]>
                    <div
                        className=[%  if message.userId == LoggedInUser.id %]
                                                { "p-2 rounded-lg max-w-lg bg-blue-100" }
                                            [% else %]
                                                { "p-2 rounded-lg max-w-lg bg-gray-100" }
                                            [% endif %]>
                        <p>{message.content}</p>
                        <span className="text-gray-500 text-sm">
                            {message.user.username} {" • "}
                            {message.createdAt}
                        </span>
                    </div>
                </div>
            [% endfor %]
    </div>
    )
}
```

```
component<FindMany> OnlineUsersComponent {
    findQuery: getUsers() as users,
    onError: render(
        <div>{ "An error occured" }</div>
    ),
    onLoading: render(
        <div>{ "Loading..." }</div>
    ),
    onSuccess: render(
        <div className="flex flex-col gap-2">
      <div className="font-bold text-lg mb-2">{ "Online Users" }</div>
            [% for user in users %]
                <div
          className=[% if user.id == LoggedInUser.id %]
                                            { "flex items-center gap-2 font-bold" }
                                        [% else %]
                                            { "flex items-center gap-2" }
                                        [% endif %]>
          <div
            className=[% if user.isOnline %]
                                                { "w-3 h-3 rounded-full bg-green-500" }
                                            [% else %]
                                                { "w-3 h-3 rounded-full bg-gray-500" }
                                            [% endif %]></div>
          <div>{ user.username }</div>
        </div>
            [% endfor %]
    </div>
    )
}

component<Create> CreateMessageForm  {
    globalStyle: {
        formContainer: "flex w-full"
    },
    actionQuery: createMessage(),
    formInputs: {
        content: {
            style: "flex w-full",
            input: {
                type: TextInput,
                placeholder: "Type your message here...",
                style: "flex-1 rounded-md border-gray-300 mr-2 px-4 py-2"
            }
        },
        user: {
            input: {
                type: RelationInput,
                isVisible: false,
                defaultValue: connect id with LoggedInUser.id
            }
        }
    },
    formButton: {
        name: "Send",
        style: "rounded-md bg-blue-500 text-white px-4 py-2"
    }
}

component<SignupForm> MySignupForm {
    globalStyle: {
        formContainer: "bg-white shadow-md rounded px-8 pt-6 pb-8 mb-4",
        input: "appearance-none border rounded w-full py-2 px-3 text-gray-700 leading-tight
            focus:outline-none focus:shadow-outline",
        inputLabel: "block text-gray-700 font-bold mb-2"
    },
    formInputs: {
        username: {
            label: {
                name: "Username"
            },
            input: {
                type: TextInput,
                placeholder: "Enter username"
            }
        },
        password: {
            label: {
                name: "Password"
            },
            input: {
                type: PasswordInput,
                placeholder: "Enter password"
            }
        }
    },
    formButton: {
        name: "Signup",
        style: "bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded focus:
            outline-none focus:shadow-outline"
    }
}

component<LoginForm> MyLoginForm {
    globalStyle: {
        formContainer: "bg-white shadow-md rounded px-8 pt-6 pb-8 mb-4",
        input: "appearance-none border rounded w-full py-2 px-3 text-gray-700 leading-tight
            focus:outline-none focus:shadow-outline",
        inputLabel: "block text-gray-700 font-bold mb-2"
    },
    formInputs: {
        username: {
            label: {
                name: "Username"
            },
            input: {
                type: TextInput,
                placeholder: "Enter username"
            }
        },
        password: {
            label: {
                name: "Password"
            },
            input: {
                type: PasswordInput,
                placeholder: "Enter password"
            }
        }
    },
    formButton: {
        name: "Login",
        style: "bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded focus:
            outline-none focus:shadow-outline"
    }
}

component<LogoutButton> MyLogoutButton {
    formButton: {
        name: "Logout",
        style: "w-full py-2 bg-red-500 text-white rounded-lg hover:bg-red-600"
    }
}

@route("/")
@permissions(IsAuth)
page Home {
    render(
    <div className="flex justify-center">
      <div className="flex flex-col w-full h-screen border shadow">
        <div className="flex flex-row h-full">
          <div className="bg-gray-100 flex flex-col w-1/4 p-4">
            <OnlineUsersComponent />
                        <div className="flex justify-end mt-auto">
                            <MyLogoutButton />
                        </div>
          </div>
          <div className="flex flex-col w-3/4 p-4">
            <div className="flex-1 overflow-y-auto">
              <MessagesComponent />
            </div>
            <div className="bg-gray-100 flex flex-row p-4 rounded-lg">
              <div className="flex-1">
                <CreateMessageForm />
              </div>
            </div>
          </div>
        </div>
      </div>
    </div>
    )
}

@route("/login")
page LoginPage {
    render(
        <div className="flex h-screen items-center justify-center bg-slate-800">
            <div className="space-y-2">
                <span className="text-xl text-white">{"Login form"}</span>
                <MyLoginForm />
                <a className="inline-block align-baseline font-bold text-sm text-blue-500 hover
                    :text-blue-800" href="/signup">
            { "Sign Up" }
            </a>
            </div>
        </div>
    )
}

@route("/signup")
page SignupPage {
    render(
        <div className="flex h-screen items-center justify-center bg-slate-800">
            <div className="space-y-2">
                <span className="text-xl text-white">{"Sign up form"}</span>
                <MySignupForm />
                <a className="inline-block align-baseline font-bold text-sm text-blue-500 hover
                    :text-blue-800" href="/signup">
            { "Login" }
            </a>
            </div>
        </div>
    )
}
```

Listing 9.2: Chatroom source code.

Kanban Board

```
app KanbanBoard {
    title: "Kanban Created by Ra",
    auth: {
        userModel: User,
        idField: id,
        isOnlineField: isOnline,
        lastActiveField: lastActive,
        usernameField: username,
        passwordField: password,
        onSuccessRedirectTo: "/",
        onFailRedirectTo: "/login"
    },
  clientDep: [
    ("react-beautiful-dnd", "^13.1.1"),
    ("@types/react-beautiful-dnd", "^13.1.3")
  ]
}

model User {
    id          Int        @id
    username   String    @unique
    password   String
    isOnline     Boolean     @default(false)
    lastActive DateTime  @default(Now)
    tasks      Task[]
}

model Task {
    id         Int      @id
    title      String
    user            User         @relation(userId, id)
    userId      Int
  status    String   @default("todo")
    createdAt DateTime @default(Now)
    updatedAt DateTime @updatedAt
}

@model(Task)
@permissions(IsAuth)
query<FindMany> getTasks {
    search: [ status ]
}

@model(Task)
@permissions(IsAuth)
query<Create> createTask {
    data: {
        fields: [title],
        relationFields: {
            user: connect id with userId
        }
    }
}

@model(Task)
@permissions(IsAuth)
query<Update> updateTask {
    where: id,
    data: {
        fields: [status]
    }
}

@model(Task)
@permissions(IsAuth)
query<Delete> deleteTaskById {
    where: id
}

component<Delete> TaskDeleteButton(id: Int) {
    actionQuery: deleteTaskById({
        where: id
    }),
    formButton: {
        name: "Delete",
        style: "text-red-500"
    }
}

component<Create> TaskCreateForm {
    actionQuery: createTask(),
    formInputs: {
        title: {
            input: {
                type: TextInput,
                placeholder: "Enter task name",
                isVisible: true,
                style: "border border-gray-400 rounded px-3 py-2 w-80"
            }
        },
        user: {
            input: {
                type: RelationInput,
                isVisible: false,
                defaultValue: connect id with LoggedInUser.id
            }
        }
    },
    formButton: {
        name: "Create",
        style: "bg-green-500 hover:bg-green-600 text-white px-4 py-2 ml-2 rounded"
    }
}

component<Custom> TaskComponent(index: Int, task: Task) {
  imports: [|
    import { Draggable } from "react-beautiful-dnd";
        import TaskDeleteButton from "./TaskDeleteButton";
        import { Task } from "@/types";
  |],
  fn: [|
    return (
            <Draggable key={index} draggableId={task.id.toString()} index={index}>
                {(provided, snapshot) => (
                    <div
                        className={`border p-4 rounded-lg mb-2 space-y-2 ${
                            snapshot.isDragging ? "bg-gray-100" : "bg-white"
                        }`}
                        ref={provided.innerRef}
                        {...provided.draggableProps}
                        {...provided.dragHandleProps}
                    >
                        <div className="flex justify-between items-center">
                            <div className="font-bold">{task.title}</div>
                            <TaskDeleteButton id={task.id} />
                        </div>
                        <div className="text-sm text-gray-500">
                            Created at {new Date(task.createdAt).toLocaleString()} by{" "}
                            {task.user.username}
                        </div>
                    </div>
                )}
            </Draggable>
    )
  |]
}

component<Custom> Column(columnId: String, columnName: String, tasks: Task[]) {
    imports: [|
        import { Droppable } from "react-beautiful-dnd";
        import TaskComponent from "./TaskComponent";
        import { Task } from "@/types";
    |],
    fn: [|
        if (!tasks) {
            return (
                <div >{ 'Loading...' }</div>
            )
        };

        return (
            <div
                className={`${columnId === "todo" && "bg-red-200"} ${
                    columnId === "doing" && "bg-orange-200"
                } ${columnId === "done" && "bg-green-200"} w-90  rounded-lg p-4 mr-4`}
            >
                <h3 className="font-bold mb-4">{columnName}</h3>
                <Droppable droppableId={columnId}>
                    {(provided) => (
                        <div ref={provided.innerRef} {...provided.droppableProps}>
                            {tasks &&
                                tasks.map((task, index) => (
                                    <TaskComponent
                                        key={task.id}
                                        task={task}
                                        index={index}
                                    />
                                ))}
                            {provided.placeholder}
                        </div>
                    )}
                </Droppable>
            </div>
        )
    |]
}

component<Custom> KanbanBoard {
    imports: [|
        import { useState, useEffect } from "react";
        import { DragDropContext, DropResult } from "react-beautiful-dnd";
        import TaskCreateForm from "./TaskCreateForm";
        import Column from "./Column";
        import { User, Task } from "@/types";
    |],
    fn: [|
        type TaskStatus = "todo" | "doing" | "done";

        type Tasks = {
            todo: Task[];
            doing: Task[];
            done: Task[];
        };

        const storedUser = localStorage.getItem("LoggedInUser");
        const [LoggedInUser, _] = useState<User | undefined>(
            storedUser ? (JSON.parse(storedUser) as User) : undefined
        );

        const {
            data,
            isLoading,
            error,
        } = useFetch<Task[]>({
            findFunc: Queries.getTasks,
            eventsFunc: Queries.getTasksEvents,
            model: 'task',
            accessToken: LoggedInUser.accessToken,
        });

        const [tasks, setTasks] = useState<Tasks>({
            todo: [],
            doing: [],
            done: []
        });

        function categorizeTasks(tasks: Task[]): { [status in TaskStatus]: Task[] } {
            const categorizedTasks: { [status in TaskStatus]: Task[] } = {
                todo: [],
                doing: [],
                done: [],
            };

            if (tasks) {
                for (const task of tasks) {
                    categorizedTasks[task.status as TaskStatus].push(task);
                }
            }

            return categorizedTasks;
        }

        useEffect(() => {
            if (data) {
                setTasks(categorizeTasks(data))
            }
        }, [data])

        const onDragEnd = async (result: DropResult) => {
            const { destination, source, draggableId } = result;

            if (!destination) {
                return;
            }

            if (destination.droppableId === source.droppableId && destination.index === source.
                index) {
                return;
            }

            const updateTask = Queries.updateTask({ where: draggableId })

            try {
                await fetch(updateTask, {
                    method: 'put',
                    headers: {
                        'Content-Type': 'application/json',
                        Authorization: `Bearer ${LoggedInUser?.accessToken}`,
                    },
                    body: JSON.stringify({ status: destination.droppableId }),
                });
            } catch (error) {
                console.log(error)
            }
        };

        return (
            <div className="flex flex-col items-center pt-10">
                <TaskCreateForm />
                <div className="flex justify-center pt-10">
                    <DragDropContext onDragEnd={onDragEnd}>
                        <div className="flex-grow">
                            <Column
                                columnId="todo"
                                columnName="Todo"
                                tasks={tasks.todo}
                            />
                        </div>
                        <div className="flex-grow">
                            <Column
                                columnId="doing"
                                columnName="Doing"
                                tasks={tasks.doing}
                            />
                        </div>
                        <div className="flex-grow">
                            <Column
                                columnId="done"
                                columnName="Done"
                                tasks={tasks.done}
                            />
                        </div>
                    </DragDropContext>
                </div>
            </div>
        )
    |]
}

component<SignupForm> MySignupForm {
    globalStyle: {
        formContainer: "bg-white shadow-md rounded px-8 pt-6 pb-8 mb-4",
        input: "appearance-none border rounded w-full py-2 px-3 text-gray-700 leading-tight
            focus:outline-none focus:shadow-outline",
        inputLabel: "block text-gray-700 font-bold mb-2"
    },
    formInputs: {
        username: {
            label: {
                name: "Username"
            },
            input: {
                type: TextInput,
                placeholder: "Enter username"
            }
        },
        password: {
            label: {
                name: "Password"
            },
            input: {
                type: PasswordInput,
                placeholder: "Enter password"
            }
        }
    },
    formButton: {
        name: "Signup",
        style: "bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded focus:
            outline-none focus:shadow-outline"
    }
}

component<LoginForm> MyLoginForm {
    globalStyle: {
        formContainer: "bg-white shadow-md rounded px-8 pt-6 pb-8 mb-4",
        input: "appearance-none border rounded w-full py-2 px-3 text-gray-700 leading-tight
            focus:outline-none focus:shadow-outline",
        inputLabel: "block text-gray-700 font-bold mb-2"
    },
    formInputs: {
        username: {
            label: {
                name: "Username"
            },
            input: {
                type: TextInput,
                placeholder: "Enter username"
            }
        },
        password: {
            label: {
                name: "Password"
            },
            input: {
                type: PasswordInput,
                placeholder: "Enter password"
            }
        }
    },
    formButton: {
        name: "Login",
        style: "bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded focus:
            outline-none focus:shadow-outline"
    }
}

component<LogoutButton> MyLogoutButton {
    formButton: {
        name: "Logout",
        style: "w-full py-2 bg-red-500 text-white rounded-lg hover:bg-red-600"
    }
}

@route("/")
@permissions(IsAuth)
page Home {
    render(
        <KanbanBoard />
    )
}

@route("/login")
page LoginPage {
    render(
        <div className="flex h-screen items-center justify-center bg-slate-800">
            <div className="space-y-2">
                <span className="text-xl text-white">{"Login form"}</span>
                <MyLoginForm />
                <a className="inline-block align-baseline font-bold text-sm text-blue-500 hover
                    :text-blue-800" href="/signup">
            { "Sign Up" }
            </a>
            </div>
        </div>
    )
}

@route("/signup")
page SignupPage {
    render(
        <div className="flex h-screen items-center justify-center bg-slate-800">
            <div className="space-y-2">
                <span className="text-xl text-white">{"Sign up form"}</span>
                <MySignupForm />
                <a className="inline-block align-baseline font-bold text-sm text-blue-500 hover
                    :text-blue-800" href="/signup">
            { "Login" }
            </a>
            </div>
        </div>
    )
}
```

Listing 9.3: Chatroom source code.

Chapter 10

Appendix 4: User Experiment

Questions

Introduction

1- How many years of programming experience do you have?

- None
- Less than 1 year
- 1-3 years
- 3-5 years
- More than 5 years

2- What programming languages are you comfortable working with? (Select all that apply)

- Java
- JavaScript
- TypeScript
- Python
- Rust
- Ruby
- PHP
- None

3- How would you rate your web development skills?

- None
- Beginner
- Intermediate
- Advanced
- Expert

4- What web development technologies are you familiar with? (Select all that apply)

- HTML
- CSS
- JavaScript
- None

5- What frontend frameworks are you familiar with? (Select all that apply)

- React
- Qwik
- Svelte
- SolidJS
- Angular
- jQuery
- None

6- What backend frameworks are you familiar with? (Select all that apply)

- Django
- Flask
- Ruby on Rails
- Laravel
- NodeJS/ExpressJS
- Other

7- Have you used an ORM before?

- Yes
- No

8- Are you familiar with Prisma ORM?

- Yes
- No

Syntax Comprehension Test

9- How easy is it to understand this code snippet on a scale of 1 to 5, where a higher number indicates that the task is easier?

```
model Task {
  id        Int      @id
  title     String
  isDone    Boolean  @default(false)
  createdAt DateTime @default(Now)
  updatedAt DateTime @updatedAt
}
```

Listing 10.1: User experiment - Code snippet 1.

10- In few words, please explain what did you understand (leave it empty if you did not understand anything)

11- How easy is it to understand this code snippet on a scale of 1 to 5, where a higher number indicates that the task is easier?

```
model User {
  id          Int        @id
  tasks           Task[]
}

model Task {
  id        Int      @id
  title     String
  user          User          @relation(userId, id)
  userId        Int
}
```

Listing 10.2: User experiment - Code snippet 2.

12- In few words, please explain what did you understand (leave it empty if you did not understand anything)

13- How easy is it to understand this code snippet on a scale of 1 to 5, where a higher number indicates that the task is easier?

```
model Task {
  id         Int        @id
  title      String
  categories Category[]
}

model Category {
  id    Int    @id
  name  String
  tasks Task[]
}
```

Listing 10.3: User experiment - Code snippet 3.

14- In few words, please explain what did you understand (leave it empty if you did not understand anything)

15- How easy is it to understand this code snippet on a scale of 1 to 5, where a higher number indicates that the task is easier?

```
@model(Task)
@permissions(IsAuth, OwnsRecord)
query<FindMany> getTasks {
  search: [ title, isDone ]
}

component TaskDetailComponent(task: Task) {
  render(
    <div className="flex mb-4 items-center">
      <div className="w-full text-gray-500 text-xl">{ task.title }</div>
    </div>
  )
}

component<FindMany> TaskListComponent {
  findQuery: getTasks() as tasks,
  onError: render(<div>{ "Error fetching tasks" }</div>),
  onLoading: render(<div>{ "Loading tasks..." }</div>),
  onSuccess: render(
    <>
      [% for task in tasks %]
        <TaskComponent task={ task } />
      [% endfor %]
    </>
  )
}
```

Listing 10.4: User experiment - Code snippet 4.

16- In few words, please explain what did you understand (leave it empty if you did not understand anything)

17- How easy is it to understand this code snippet on a scale of 1 to 5, where a higher number indicates that the task is easier?

```
@model(Task)
@permissions(IsAuth)
query<Create> createTask {
  data: {
    fields: [ title ],
    relationFields: {
      user: connect id with userId
    }
  }
}
```

```

component<Create> TaskCreateForm {
  actionQuery: createTask(),
  formInputs: {
    title: {
      input: {
        type: TextInput,
        placeholder: "Enter task title"
      }
    },
    user: {
      input: {
        type: RelationInput,
        isVisible: false,
        defaultValue: connect id with LoggedInUser.id
      }
    }
  },
  formButton: {
    name: "Create"
  }
}
```

Listing 10.5: User experiment - Code snippet 5.

18- In few words, please explain what did you understand (leave it empty if you did not understand anything)

19- How easy is it to understand this code snippet on a scale of 1 to 5, where a higher number indicates that the task is easier?

```
@model(Task)
@permissions(IsAuth, OwnsRecord)
query<Delete> deleteTaskById {
  where: id
}

component<Delete> TaskDeleteButton(id: Int) {
  actionQuery: deleteTaskById({
    where: id
  }),
  formButton: {
    name: "Delete"
  }
}
```

Listing 10.6: User experiment - Code snippet 6.

20- In few words, please explain what did you understand (leave it empty if you did not understand anything)

Developer Experience Test

21- In few words, can you explain why the compiler is raising this error and how it can be fixed? (Leave it empty if you don't know)

Compiler error

UndefinedError("@(Line:15): Undefined field 'name' in Model 'Task' ") "In few words, can you explain

```
model Task {
    id        Int      @id
    title     String
    isDone    Boolean  @default(false)
    createdAt DateTime @default(Now)
    updatedAt DateTime @updatedAt
}

@model(Task)
@permissions(IsAuth)
query<Create> createTask {
  data: {
    fields: [ title, name ],
    relationFields: {
      user: connect id with userId
    }
  }
}
```

Listing 10.7: User experiment - Code snippet 7.

22- In few words, can you explain why the compiler is raising this error and how it can be fixed? (Leave it empty if you don't know)

Compiler error

TypeError("@(Line:20): Excepted a value of type 'Task' but received 'task' of type 'String' in 'TaskDetailComponent' ")

```
1  component TaskDetailComponent(task: Task) {
2    render(
3      <div className="flex mb-4 items-center">
4        <div className="w-full text-gray-500 text-xl">{ task.title }</div>
5      </div>
6    )
7  }
8
9  component<FindMany> TaskListComponent {
10   findQuery: getTasks() as tasks,
11   onError: render(<div>{ "Error fetching tasks" }</div>),
12   onLoading: render(<div>{ "Loading tasks..." }</div>),
13   onSuccess: render(
14     <>
15       [% for task in tasks %]
16         <TaskComponent task={ "task" } />
17       [% endfor %]
18     </>
19   )
20 }
```

Listing 10.8: User experiment - Code snippet 8.

23- In few words, can you explain why the compiler is raising this error and how it can be fixed? (Leave it empty if you don't know)

Compiler error

FormInputTypeError("@(Line:6): Expected an input of type 'TextInput' for field 'title' but an input of type 'NumberInput' was implemented instead in 'TaskCreateForm' ")

```
1  component<Create> TaskCreateForm {
2    actionQuery: createTask(),
3    formInputs: {
4      title: {
5        input: {
6          type: NumberInput,
7          placeholder: "Enter task title"
8        }
9      },
10     user: {
11       input: {
12         type: RelationInput,
13         isVisible: false,
14         defaultValue: connect id with LoggedInUser.id
15       }
16     }
17   },
18   formButton: {
19     name: "Create"
20   }
21 }
```

Listing 10.9: User experiment - Code snippet 9.

24- In few words, can you explain why the compiler is raising this error and how it can be fixed? (Leave it empty if you don't know)

Compiler error

TypeError("@(Line:4): Excepted a value of type 'String' but received 'false' of type 'Boolean' in '**default?**' ")

```
 1  model Task {
 2    id         Int      @id
 3    title      String
 4    isDone     String   @default(false)
 5    user           User         @relation(userId, id)
 6    userId         Int
 7    createdAt DateTime @default(Now)
 8    updatedAt DateTime @updatedAt
 9  }
```

Listing 10.10: User experiment - Code snippet 10.

25- In few words, can you explain why the compiler is raising this error and how it can be fixed? (Leave it empty if you don't know)

Compiler error

UniqueFieldError("@(Line:14): Excepted a field of type 'UniqueField' but received 'title' of type 'NonUniqueField' in 'deleteTaskById' ")

```
 1  model Task {
 2    id         Int      @id
 3    title      String
 4    isDone     Boolean  @default(false)
 5    createdAt DateTime @default(Now)
 6    updatedAt DateTime @updatedAt
 7  }
 8
 9  @model(Task)
10  @permissions(IsAuth, OwnsRecord)
11  query<Delete> deleteTaskById {
12    where: title
13  }
```

Listing 10.11: User experiment - Code snippet 11.

Reflection

26- How do you feel about the syntax on a scale of 1 to 5, where a higher number indicates that the bigger the better?

27- Were there any particular aspects of the DSL syntax that you found challenging or confusing?

28- Were there any aspects of the DSL that you feel could be improved or made more user-friendly?

29- On a scale of 1 to 5, where a higher number indicates that the bigger the better, how helpful were the compiler messages in identifying and fixing bugs in the DSL code?

30- Which of the following compiler messages that you found particularly helpful or informative? (Select all that apply)

- UndefinedError
- TypeError
- UniqueFieldError
- FormInputTypeError

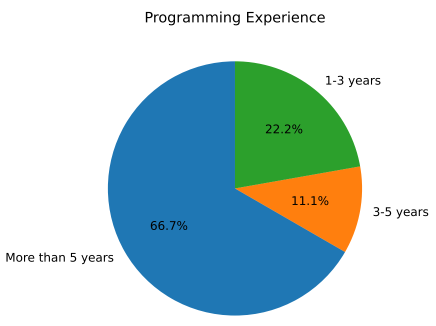
31- Would you use the DSL or similar tool for personal projects?

- Yes
- No
- Maybe

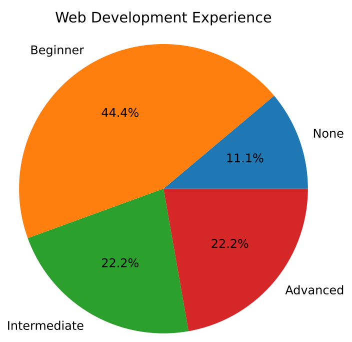
32- What size would this DSL be helpful for?

- Small projects
- Medium projects
- Large projects
- None

Results

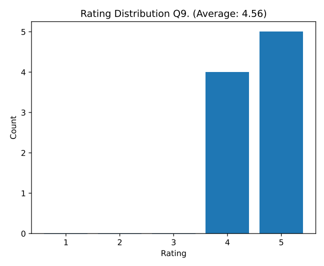


(a)

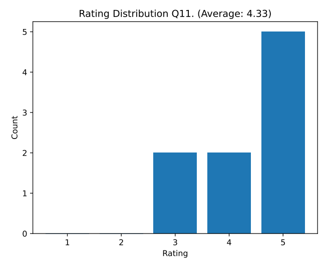


(b)

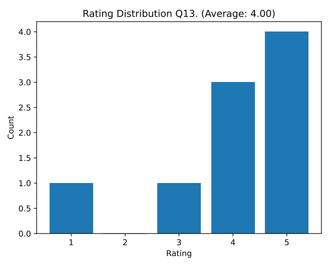
Figure 10.1: Participants' programming and web experience.



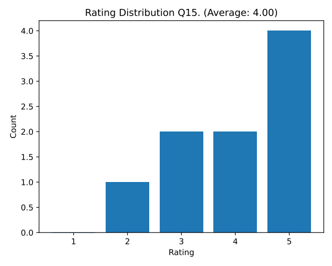
(a)



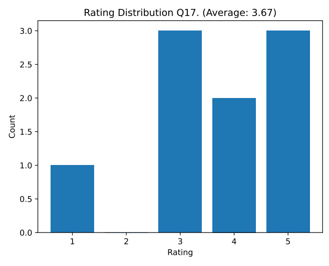
(b)



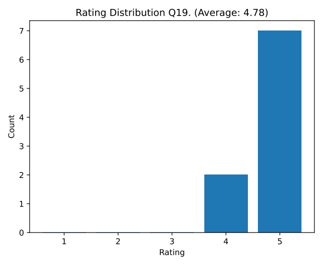
(c)



(d)



(e)



(f)

Figure 10.2: The rating distribution of each code snippet, as rated by the participants.

Question No.	Participant A (25 min)	Participant B (60 min)	Participant C (30 min)
1	More than 5 Years	More than 5 years	More than 5 years
2	Python	Java, JavaScript, Python, Rust	Java, JavaScript, Python, TypeScript
3	None	Beginner	Beginner
4	HTML, CSS, JavaScript	HTML, CSS, JavaScript	HTML, CSS, JavaScript
5	None	jQuery	React
6	NodeJS/ExpressJS	Django, Flask	Django, NodeJS/ExpressJS
7	No	No	No
8	No	No	No
9	5	4	4
10	an entity called Task that has properties (unique id, title, is done? property, the time it is updated and created)	It defines the data model for a task which is described by five fields. The first column is the field's name, the second column is its data type, and the third is for extra attributes. @default specifies a field's default value. @id specifies that a field is a unique identifier/primary key. I'm not sure what @updatedAt does.	A Task model is being defined with 5 attributes (eg. id). Each attribute has an expected datatype. For example, the value of isDone is expecated to be a boolean, true or false. Some of the attributes are initialised with a default value if one isn't provided at the time of instantiating this model. For example, the value of isDone will be set to false if a value isn't provided.
11	5	5	3
12	Same as the previous question but has user assigned to it	This defines two entities User and Task. A User has an integer ID which is its primary key and an array of tasks. A task has an integer ID, a string title, a user ID and a reference to the User with that ID. @relation describes a relation using a foreign key and the primary key it relates to. In other words there is a one-to-many relation from User to Task.	Two model/object definitions. One is of type User, one is of type Task. The User model has an attribute/property called tasks and it is expected to be an array of elements that are of type Task. The Task model has a property called user which is of type User and declares a relationship between its own userId property and the User model's id property.
13	5	5	4
14	Same as the previous but has a category and the category has its own properties	This describes two entities Task and Category which have a many-to-many relationship. Both entities have an integer ID and a string title/name. A Task has an array of categories and a Category has an array of tasks.	Two models/objects being defined with a set of properties and those properties expected datatypes. The Category object accepts an array of Task objects for the tasks property and the Task object accepts an array of Category objects for the categories property.
15	2	3	5

Continued on next page...

16	It is getting all the tasks, then visualizing the tasks	getTasks is a query that searches for many Tasks according to the tasks' 'title' and 'isDone' fields. It has the IsAuth and OwnsRecord permissions. TaskDetailComponent is a UI element that takes a task as a parameter, and displays a its title inside some formatted <div>s. TasksListComponent is a UI element that uses the getTasks query to obtain a list of tasks. While the query is loading a message is displayed. If the query succeeds each task is displayed in a TaskDetailComponent. If an error occurs an error message is displayed.	getTasks: A query to get tasks is defined to extract and store the title and the value of isDone for all the tasks available in the record. It can be executed on the Task model. The requirements for this to be allowed involve the end user's account being authenticated and owning a record. TaskDetailComponent: A definition of a component that takes in an object of type Task and consists of two nested div blocks. The outer one centres the inner div. The inner div allows the title of the task to be rendered in an extra large grey font. TasksListComponent: The returned values from the getTasks query are stored as a variable called tasks. If there is an error while executing the getTasks query, a div block is rendered on the page, informing the user that there was an "Error fetching tasks". While attempting to retrieve the tasks, the div block is rendered with the "Loading tasks" text. If the getTasks query is executed completely with no issues, a list of TaskDetailComponents are rendered on the page based on the output of the query where each TaskDetailComponent displays information about each task in the list.
17	1	3	5

Continued on next page...

18	N/A	createTask is a query that creates a Task by providing its title and relating its userId to a user. TaskCreateForm is a UI component that is a form which triggers the createTask query when submitted. It has a text input field for the task's title and a hidden input for the user's ID, as well as a button to submit the form. The title input has some placeholder text. The user input uses the ID of the logged-in user.	The end user must have an authenticated account for the following code to be executed. It allows the end user to fill in a form where they can create a task
19	5	4	5
20	Implemeting a delete tas functionality in UI with query	deleteTaskById is a query for deleting the Task that has a particular ID. TaskDeleteButton is a UI element that is parameterised by an integer ID. It defines a button with the text ”Delete” which triggers the deleteTaskById query using the given ID.	User must be logged in and have an existing record. The code allows the user to delete a task through the use of a delete button
21	there is no name property in Task Model	The createTask query specifies a ’name’ field for a Task, but Tasks do not have a field called ’name’. You could remove ’name’ from line 15 so that it only contains ’title’, or add a field called ’name’ to Task.	The name field does not exist in the Task model. This needs to be defined in the model with a datatype (eg String).
22	it should not be inside ”” it should be only the variable task	Instead of providing the ’task’ variable to each TaskDetailComponent, the string literal ”task” is used. Remove the double quotes in line 20.	The TaskDetailComponent component expects a Task object being passed in. Line 20 passes in a string. The value, task, can be passed in without speech marks to solve the problem.
23	The title should have TextInput instead of the NumberInput	The ’title’ input field is of type NumberInput but should be a TextInput. Replace NumberInput in line 6 with TextInput.	The title field should be a string and therefore the user input should be of type TextInput rather than NumberInput as this would mean the title is of the wrong type for the model
24	As its default is boolean not string as it is defined type	The boolean value ’false’ was provided as a default value for the String field ’isDone’. Replace ’String’ with ’Boolean’ in line 4.	The default value and expected datatype for the isDone field are contradicting each other. The datatype should be changed to Boolean to solve the problem.

Continued on next page...

25	as title is not unique it should have been id instead	The query deleteTaskById is deleting according to a Task's 'title' field, which is not a unique field. Replace 'title' in line 14 with 'id'.	The id should be used rather than the title as this is a unique identifier
26	4	4	4
27	not really, maybe the part of ui was more challenging	It was not clear what some of the attributes (@something) do. The angle-bracket notation for different types of query was not clear at first. It resembles type parameters in C++ or Rust but I'm unsure whether 'Create', 'FindMany' and 'Delete' are types.	I only realised at the last question what the significance of the @ symbol was when defining the model.
28	maybe the closeness of the language to the human language	The language itself seems like it would be very usable after taking some time to learn it. Because it uses type inference, an editor plugin that enables inspecting the inferred types would be very helpful.	The syntax in general is really good and I could understand most of it. No improvements needed
29	5	5	5
30	UndefinedError, TypeError, FormInputTypeError, UniqueFieldError	UndefinedError, TypeError, FormInputTypeError	UndefinedError, TypeError, FormInputTypeError, UniqueFieldError
31	Yes	Yes	Maybe
32	Medium projects	Medium projects	Medium projects

Question No.	Participant D (31 min)	Participant E (46 min)	Participant F (22 min)
1	More than 5 Years	More than 5 Years	More than 5 years
2	Java, JavaScript, Python	Java, JavaScript, TypeScript, Python	TypeScript, JavaScript, PHP, Python
3	Intermediate	Advanced	Advanced
4	HTML, CSS, JavaScript	HTML, CSS, JavaScript	HTML, CSS, JavaScript
5	React, Angular, jQuery	React, Angular, jQuery	React, Angular, jQuery
6	Flask, NodeJS/ExpressJS	Django, Flask, NodeJS/ExpressJS	Laravel, NodeJS/ExpressJS, Flask
7	No	Yes	Yes
8	No	Yes	Yes
9	5	4	5
10	it defines a model called Task which has 5 properties and of designed types. Id is of type Int and is unique. Isdone and createat have respectively default values when created and not assigned.	This is a data model representing a Task. The Task has the field id which is an integer and is also the Task's unique identifier. The Task also has the fields title, which is a string, isDone which is a boolean defaulted to false (so when no value is provided, it will be false), createdAt which is a datetime defaulted to the time of the Task object's instantiation, and updatedAt which is a datetime that is automatically updated every time the Task object is updated.	I understand that the snippets above defines a database model, where 'id' field is an integer primary key, 'title' is a string field, 'isDone' is a boolean field with a default value of false, 'createdAt' is a timestamp field with default value as the current server's timestamp, I guessed that'@updatedAt' notation maps the field to the time of the most recent update.
11	4	5	5

Continued on next page...

12	2 models. Similar to previous one. I don’t understand what the parameters for @relation are, weather the id for the parameters refers to task or user. Also why wr need userid and user entity belongs to task together. The info for user and userid should be defined separately if considering database / software design?	The code snippet shows User and Task models. The User has an id field which is an integer, and is the User model’s unique identifier. User also has a tasks field which is an array of the Task objects. The Task model has an id field which is an integer, and is the Task model’s unique identifier. Task also has a title field which is a string, user field which is actually a relation representing a User object, and the relation’s id (userid, which is a foreign key). This establishes one-to-many relationship, where one User can have many Tasks, but one Task can only be tied to one User.	‘User‘ model has a one to many relationship with ‘Task‘ model, the foreign key on ‘Task‘ is ‘userId‘ field.
13	1	5	4
14	There’s a definition loop	The code snippet shows Task and Category models. The Task model has an id field which is an integer, and is the Task model’s unique identifier. Task also has a title field which is a string, and a categories field which is an array of Category objects. The Category model has an id field which is an integer, and is the Category model’s unique identifier. Category also has a name field which is a string, and a tasks field which is an array of Task objects. I can interpret from this code snippet that this establishes a many-to-many relationship between Task and Category, where one Task can have many Categories, and one Category can have many Tasks.	A many-to-many relationship. It’s a bit confusing, but I believe it implies that the schema parser would automatically create pivot/join table and connect records?
15	5	5	4

Continued on next page...

16	Reads tasks as list and use each data to render a component for all data. List rendering deals with error etc cases.	TaskCreateForm is a component that neatly represents a form. This form has title and user inputs. This form is used to create a task. The input type for the title is TextInput, meaning just text. The input type for the user is RelationInput. The user input is not visible, as the user does not manually enter it, instead it is automatically filled using the logged in user's ID. When the form is submitted, by clicking the button that says "Create", it invokes the createTask() query, specified by actionQuery in the form component. The createTask query then uses the inputs to the form to create a Task record. The user that clicks 'create' can only create a Task record using this form if they are authenticated.	Seems like a field relevant to Task database Model. I assume the permission annotation would only make it accessible for this who created the record. FindMany query definition at the top is a bit confusing, I am not sure if search defines searchable items or selected fields. TaskListComponent seems to be a data fetching handler and view manager for TaskList, I expect that findQuery is the data source, onError, onLoading, onSuccess is for handling views in different data fetching states.
17	4	5	5
18	Renders a form but I don't know what's the connect syntax's meaning	TaskCreateForm is a component that neatly represents a form. This form has title and user inputs. This form is used to create a task. The input type for the title is TextInput, meaning just text. The input type for the user is RelationInput. The user input is not visible, as the user does not manually enter it, instead it is automatically filled using the logged in user's ID. When the form is submitted, by clicking the button that says "Create", it invokes the createTask() query, specified by actionQuery in the form component. The createTask query then uses the inputs to the form to create a Task record. The user that clicks 'create' can only create a Task record using this form if they are authenticated.	Same as the previous answer, on model and permission fields. The create query creates a task and links it to the authenticated user. 'TaskCreateForm' is defining the query that's to be executed on form submission, and the inputs to show in the frontend form.
19	5	5	5

Continued on next page...

20	Renders a button and defines the delete JH behaviour for button	TaskDeleteButton is a component that simply represents a button. This button reads ”Delete”, and when clicked it invokes the deleteTaskById() query, specified by actionQuery inside the component. The deleteTaskById query takes a single ’where’ argument, which specifies the ID for the Task record to delete. The user that clicks ’delete’ can only delete the Task record if they are authorised and own the Task record that they are trying to delete.	Pretty much the same as the pervious one, but instead of Creation, it handles deletion form and query.
21	name probably should be replaced by title	The error arises because there is no ’name’ field in the Task model. This can be fixed by removing ’name’ from the fields in the createTask query, or by creating a ’name’ field in the Task model.	The createTask query has a field that doesn’t exist inside the ‘Task‘ model, can be fixed by either adding a ‘name‘ field to ‘Task‘ model or removing ‘name‘ from data fields.
22	Probably should remove “” for task	The error arises because you are passing the task prop to the TaskDetailComponent as a string. It is specified in the signature of TaskDetailComponent that the task prop should be of type Task, and not of type String. To fix this, remove the quotes around ”task” on line 20.	The value passed to TaskDetailComponent doesn’t match the expected type, can be fixed by replacing ”task” with task.
23	Change numberinput to TextInput	The error arises because you have put the input type for the ’title’ field in the form to be a NumberInput. To fix this, replace NumberInput with TextInput, because the title field in the Task model is a String not a Number/Integer.	The form input type is set to NumberInput, the ‘Task‘ database model maps this field to ‘String‘. This can be fixed by using ‘StringInput‘ (if available idk) on line 6

Continued on next page...

24	Change the type to Boolean or false to some string value	This error arises because you are passing a boolean default value (false) to a String field. To fix this, isDone should be of type Boolean not of type String. Alternatively, if you want isDone to be of type String then the default value should also be a String, e.g. ”” (double empty quotes).	The default value for a string field is set to a boolean value. This can be fixed by replacing ‘String‘ value with ‘Boolean‘ on line 4
25	Change title to id	This error arises because you are using ’title’ as the where clause for selecting a task to delete in the deleteTaskById query. The title field is not a unique identifier for the Task model, and as such two Task records could have the same title field value, causing uncertainty when deleting on ’where title’. To fix this, you should replace ’title’ on line 14 with ’id’. This is because id is the unique identifier for the Task model, highlighted by the @id attribute on line 2.	The delete query is depending on a non-unique field, which means that it can delete multiple records by mistake. Can be fixed by replacing title with id on line 14
26	4	5	4
27	N/A	The <> syntax was a little confusing, but this is because I am not so confident with generics in general. I am assuming these are similar to generics, because of the <> syntax which represents generics in TypeScript. However, it did not impede my ability to understand the code syntax. In fact, in many cases it helped.	With the delete query it was a bit confusing how the DSL differs from DELETE and DELETE many.
28	Can’t think of	It would be good to have some documentation that describes the meaning of the values in the <> syntax, such as the different query types. Other than that, this syntax appear very easy to use and understand.	N/A
29	5	5	5
30	UndefinedError, TypeError, FormInputTypeError, UniqueFieldError	UndefinedError, TypeError, FormInputTypeError, UniqueFieldError	UndefinedError, TypeError, FormInputTypeError, UniqueFieldError

Continued on next page...

31	Maybe	Yes	Maybe
32	Medium projects	Large projects	Small projects

Question No.	**Participant G (34 min)**	**Participant H (64 min)**	**Participant I (50 min)**
1	1-3 years	1-3 years	3-5 years
2	Java, Python	JavaScript, TypeScript, Python, PHP	Python, Rust
3	Beginner	Intermediate	Beginner
4	HTML, CSS	HTML, CSS, JavaScript	None
5	None	React, jQuery	None
6	Django	Laravel, NodeJS/ExpressJS	Django
7	No	Yes	Yes
8	No	No	No
9	4	5	5
10	A Task has 5 attributes (id, title, isDone, createdAt and updatedAt) each attribute is a different type, e.g id in an integer, title is string etc.	It is table for tasks that indicates if a task is done or not. Also, it specify its title and when it is added and updated.	This code defines a table within a database for storing tasks. Each task has a title, whether it is completed (false by default), when it was created (now by default) and when it was updated (automatically incremented on each update). The 'id' column is the primary key for each entry in this table
11	4	5	3
12	Each user has an id and an array of tasks, each task has an id, title and a user that its connected to	one record in a table User can be associated with one or more records in table Task	There is a User table and a Task table within the database. Each task has a user relation, and a user can have multiple tasks related to them. What I don't properly understand is why there is a 'userId' and a 'user' in the Task model
13	3	5	4
14	A task can have an id, title and be a part of many categories. A category has an id, a name and an array of tasks which are part of it	one or multiple records in a table Task can be associated with one or multiple records in table Categories.	This represents a many-to-many relationship between tasks and categories.
15	3	5	4

Continued on next page...

16	I think it is trying to fetch all the tasks in the database and rendering it onto the webpage if successful.	fetch the done tasks and list them by title	The TasksListComponent executes the getTasks query, returning the 'title' and 'isDone' for all tasks owned by the user visiting the component. The TasksListComponent also has separate HTML defined for loading a page and if there is any error. On success of the query, the TasksListComponent renders the TaskDetailComponent for each of the user's tasks.
17	3	4	3
18	I think this is creating the page in which the user can create a task where they enter a task title which is connected to that user. then they press 'create' button to create it	create a new task and link it by the logged in user	The TaskCreateForm takes as input a task title, as well as the user (but this is grabbed automatically based on who is logged in). The createTask query is executed with the user and title, to create a task with a user. The task is related to the user based on 'connecting' ids but I don't properly get this
19	4	5	5
20	Deleting the task with the corresponding id. User can press 'Delete' button to delete it	just delete a task	This defines a TaskDeleteButton component that takes as input an ID, and executes a query to delete the Task with that id, provided the user owns that record. This button is also labelled with 'Delete'.
21	'name' is not a valid attribute name in the Task model	Task model is created in a wrong way	In the createTask query, there is a reference to a 'name' field in Task, but this field does not exist in the Task model. To fix this, 'name' should be replaced with 'user'.
22	Here, "task" is a string but it needs to be of type task. Get rid of the quotation marks	task is passed as a string not as a variable	The TaskDetailComponent has been given the string "task" when it should have been passed a task object. To fix this, remove the quotation marks from "task" on line 20.
23	The type in line 6 should be TextInput instead of NumberInput as the user is entering a title for the task	title should be string	The Task title should be text instead of a number. To fix this, replace 'NumberInput' with 'TextInput' on line 6.

Continued on next page...

24	In line 4, the isDone attribute should be of type Boolean instead of string	isDone should be Boolean	isDone is defined to be a String type, but has been given a boolean default value. To fix this, either set the default value to be a string, such as ”false”, or better, change isDone to be a Boolean column.
25	title is not if type ’UniqueField’, should be id instead	title is not unique field it should be replaced by id for example	The deleteTaskById query requires a unique field from the Task model. ’title’ is not unique. To fix this, provide a unique field on line 14. In this case, we should replace ’title’ with the id field on line 14
26	4	4	4
27	wasn’t too sure what the @ signs meant but Im not very good at programming so could just be me	NO	I found the definition of relations to sometimes be confusing. The many-to-many syntax with [] square brackets was great and very simple!! But I was confused by the foreignkey ’connect’ syntax for ids, as well as when the Task model required both a ’user’ AND a userId to be specified in its model definition, in order to be linked to a User. At first I was confused a bit by the scope of some variables. For instance, some of the components referred to things such as ’title’ and ’task’, when they were never provided as inputs to the component. Although I did eventually understand that these were defined globally in the file.
28	no	N/A	Perhaps the syntax for ’connecting’ ids as mentioned before, or at least it would need clear documentation.
29	5	4	5
30	UndefinedError, FormInputTypeError	UndefinedError, TypeError, FormInputTypeError, UniqueFieldError	UndefinedError, TypeError, FormInputTypeError, UniqueFieldError
31	Yes	Yes	Maybe
32	Small projects	Medium projects	Medium projects

References

[1] “TypeScript-first schema validation with static type inference.” https://github.com/colinhacks/zod.

[2] E. Cooper, S. Lindley, P. Wadler, and J. Yallop, “Links: Web programming without tiers,” in *Formal methods for components and objects*, 2007, pp. 266–296.

[3] *The opa language for web application development.* 2011 [Online]. Available: http://opalang.org/

[4] M. Serrano, *Hop, multitier web programming.* Inria, 2006 [Online]. Available: http://hop.inria.fr

[5] *Wasp: Develop full-stack web apps without boilerplate.* 2020 [Online]. Available: https://wasp-lang.dev/

[6] Stack Overflow, “Stack overflow developer survey 2022.” 2022 [Online]. Available: https://survey.stackoverflow.co/2022/#most-popular-technologies-webframe

[7] “Prisma schema API (Reference).” https://www.prisma.io/docs/reference/api-reference/prisma-schema-reference.

[8] Windi CSS Team, “Windi CSS.” https://windicss.org/.

[9] “Jinja.” https://jinja.palletsprojects.com/en/3.1.x/.

[10] Atlassian, “GitHub - atlassian/react-beautiful-dnd: Beautiful and accessible drag and drop for lists with React.” https://github.com/atlassian/react-beautiful-dnd, 2022.

[11] V. Markan, “React SEO Best Practices and Strategies.” https://www.toptal.com/react/react-seo-best-practices.

[12] “JSX.” https://facebook.github.io/jsx/.