Modern Web Development with Deno

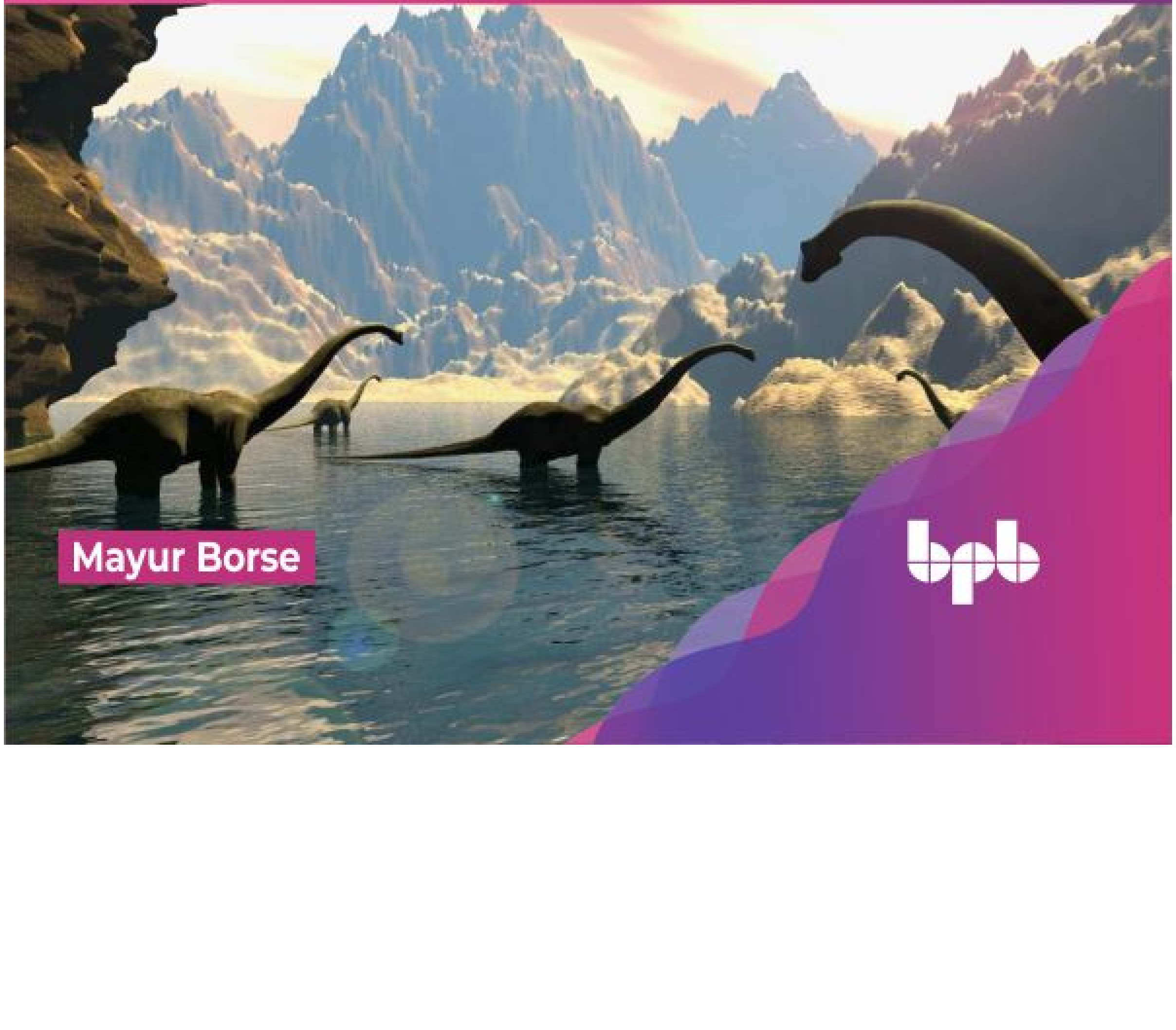
Develop Modern JavaScript and TypeScript Code with Svelte, React, and GraphQL

Mayur Borse

bpb

Modern Web Development with Deno

Develop Modern JavaScript and TypeScript Code with Svelte, React, and GraphQL



Mayur Borse

bpb

Modern Web Development with Deno

Develop Modern JavaScript and TypeScript
Code with Svelte, React, and GraphQL

Mayur Borse

www.bpbonline.com

Group Product Manager: Marianne Conor

Publishing Product Manager: Eva Brawn

Senior Editor: Connell

Content Development Editor: Melissa Monroe

Technical Editor: Anne Stokes

Copy Editor: Joe Austin

Language Support Editor: Justin Baldwin

Project Coordinator: Tyler Horan

Proofreader: Khloe Styles

Indexer: V. Krishnamurthy

Production Designer: Malcolm D'Souza

Marketing Coordinator: Kristen Kramer

First published: 2023

Published by BPB Online

WeWork, 119 Marylebone Road

London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55510-969

www.bpbonline.com

Dedicated to

"*To all the Developers in the world*"

About the Author

Mayur is a Dynamic and Result Oriented professional with a passion for Software Engineering and Perfection. Currently, working as a Full Stack Software Engineer on various projects utilising Gatsby.js, Nikola, HTML,CSS, Bootstrap, Javascript and so on. Mayur has also used various design patterns such as CRUD, SearchBar, CSVDownloader, Loader for UI creation of a network device and written blogs on each of them. He has done Bachelors of Electronic Engineering from University of Pune, along with a Diploma in Electronics and Telecommunications. Apart from this, he posseses 60+ certifications on Pluralsight, which includes course subjects such as React, React Native, GraphQL, Python, Django, Javascript, etc. He is also passionate about philosophy and spirituality.

About the Reviewer

Chad Elofson is a Software Analyst using Node and Deno-based technologies. He works at Thompson Rivers University as a Software Analyst. He started his IT career 19 years ago as Network Administrator. For the past eight years, he has moved to Software Engineering focused on JavaScript runtime-based technologies.

Acknowledgement

I’m grateful to the team at BPB Publications for giving me the opportunity to write the book.

Preface

This book introduces developers to the new JavaScript runtime named Deno. It also shows how to create web applications on the front-end and the back-end with TypeScript using Deno.js.

This book takes a practical approach for web developers or JavaScript. It covers a few realtime examples as well.

This book is divided into **9 chapters**. They will cover Introduction to Deno, Introduction to TypeScript, Create Web Applications with React, Introduction to GraphQL, Creating GraphQL API server, Creating Svelte Application etc. The details are listed below.

Chapter 1 will cover what Deno is, why Deno was introduced by the creator of Node.js (popular JavaScript runtime), what are the differences between Deno and Node etc. Will list features of Deno. Will explain different technologies used in creating Deno. It will also provide information about What is JavaScript runtime and What is JavaScript engine.

Chapter 2 will cover What is TypeScript, benefits of TypeScript, features of TypeScript etc. It will also explain the basics of TypeScript for the JavaScript developers. It will also provide details about how to set up TypeScript.

Chapter 3 will provide details on how to install Deno. It will also list some of the widely used developer tools like VS Code, Terminal etc. necessary for development.

Chapter 4 will cover details about the Deno Ecosystem. Deno ecosystem includes Deno CLI details like Environment variables, subcommands, how to pass arguments, File watcher, Permission flags, Permission allow-list etc. It will also list Deno runtime APIs like Web APIs, Global APIs etc.

Deno modules which are an essential part of the ecosystem are listed as two categories Standard library and Third-party modules. Also, Module registries like Deno's own site deno.land/x and nest.land will also be listed.

Chapter 5 will introduce Aleph.js, A Full-stack React framework for Deno. It will list the concepts like pages, routing etc. in brief. It will create a Todos App API using provided API like GET, POST, DELETE etc. This will be helpful for the developer to get familiar with the concepts utilized in the next chapter.

Chapter 6 will continue the Todos app development and will cover UI implementation. It will provide details on how to use the Styled component library to create React components with CSS styling. It will provide code to create basic components like Button, Input etc. and will utilize them in the app along with TodoList and TodoDetails components.

Chapter 7 will introduce GraphQL and will provide details about features like precise fetching, Single API Endpoint etc. It will also list some of the core concepts like Type, Input, Mutation, Resolver etc. It will then introduce oak module along with oak-graphql which will be used to implement GraphQL Server in the next chapter.

Chapter 8 will cover implementation of GraphQL server using oak, oak-graphql for creating Phonebook app. It will provide details on how to create a git repo, will use trex as package manager and velociraptor as script runner. It will also provide the code for server setup, Schema, Resolvers. Database setup using ORM and authentication etc.

Chapter 9 will introduce the promising JavaScript framework or compiler Svelte. Snel is a module for creating Svelte applications using Deno. It will cover the setup process for Snel and snel-carbon (UI component module). It will then provide code for setting up public private routes for the app along with Svelte components.

Table of Contents

CHAPTER 1

The Deno Land

Introduction

After a decade of success of **Node** runtime, its creator *Ryan Dahl* has come up with another exciting technology called **Deno**, which is also a runtime like Node but is more secure and provides many improvements and modern features in comparison to Node. Dahl stormed the development world by announcing **Deno** during a *JSCONF EU* talk in 2018 named *10 things I Regret about Node.js* (**https://www.youtube.com/watch?v=M3BM9TB-8yA**). Since then, Deno has been a topic of excitement in the software industry.

Structure

In this chapter, we will cover the following topics:

- Introducing Deno
    - The Node revolution
- Comparison of Deno and Node
- Deno features
- Deno foundation stack
- What is a JavaScript engine?
- What is a JavaScript runtime?

Objectives

After studying this chapter, you will be familiar with the main features of Deno and its internal architecture. You will also be able to understand the reasons for the creation of Deno, the role of Node in the creation of Deno, the similarities and differences between Deno and Node, and the stack used for creating Deno, that is, Chrome V8 Engine by Google, Rust by Mozilla, and Tokio.

You will also be familiar with how the JavaScript engine works and how the JavaScript runtime works.

Introducing Deno

Deno is a secure and modern JavaScript runtime that uses the V8 engine for JavaScript code conversion, Tokio (written in Rust) for async event handling and the API backend written in Rust. It provides features as follows:

- Integrated security. Access to the network, file, and environment is provided only through flags.
- Built-in tools such as formatter, test-runner, debugger, linter, and so on.
- Typescript support out of the box.
- Top-level await (without async).
- ES Modules import (default).
- No package manager is required as module imports are done using URLs.

Deno is the creation of *Ryan Dahl*. Interestingly, Dahl is also the creator of Node, which is also a runtime. In fact, Deno was created to overcome the limitations of Node and provide modern features out of the box, which is not part of Node runtime. Hence, the journey of Deno begins with Node, which changed the way JavaScript was perceived and used in the industry. It is a fact that JavaScript has become a widely used and revered language after the entry of Node.

Many developers are using only JavaScript for their entire stack, thanks to Node. Therefore, to understand “*The Deno Evolution*”, one must begin with “*The Node Revolution*”.

The Node Revolution

A decade ago, JavaScript was mocked as a funny and peculiar language useful mostly for front-end Web development. The scope of JavaScript was very limited compared to mainstream languages like C++ or Java, as it was mainly useful for browser-based applications only.

In 2009, *Ryan Dahl* introduced Node.js to the world. **Node** is a JavaScript runtime written in C++ which uses V8 Engine by Google (launched in 2008) and **libuv** event loop to perform asynchronous operations in the background. Since its release, Node.js has revolutionized JavaScript usability and is being used by several tech giants such as PayPal, LinkedIn, Netflix, and Uber to name a few. Let us see the features that led to the rise of Node.

JavaScript everywhere

Node has been the reason that made JavaScript available outside the browser. Nowadays, JavaScript is used for development across many domains. **Electron.js** is used for cross-platform desktop apps, **React.js** for Web development, and **React Native** for mobile development. Many companies and developers use JavaScript alone for their entire stack as the code logic can be moved from client to server because both are using the same language.

Asynchronous and event-driven architecture

Node is single-threaded, meaning it contains only one call stack that executes only one program at a time. It has event-driven architecture. This means executions in Node are done according to some event occurrence. Node uses an event loop **libuv** to handle executions asynchronously, i.e., in a non-blocking way.

Open-source and cross-platform

Node is open source which is one of the major factors for its popularity and acceptance amongst the developers.

Cross-platform support is also an amazing feature node offers to developers. It works on OSX, Linux and Windows very well. Using this benefit, a developer can write node code on OSX and can use Linux to deploy.

Very performant

Node uses V8 JavaScript Engine under the hood. The V8 engine is very fast, and this makes the Node a very high-performing runtime. Performance is one of the major feature node offers.

A node package manager (npm) for handling dependencies

Node package manager **`npm`** is used to install packages and manage package dependencies. It is also a centralized registry where users can publish their node packages.

Event loop

Node uses **libuv** event loop for performing async i/o operations. The event loop uses workers and a thread pool for performing operations in the background. It also checks if the call stack is empty and the main programme has been executed so it can push the code from the callback queue to the empty call stack. The JavaScript runtime process is available in visualization here (**http://latentflip.com/loupe/**). We will check the JavaScript runtime architecture in detail later in this chapter.

Along with such great features, Node has several shortcomings, which led its creator *Ryan Dahl* to start a new project (Deno). In the next section, we will compare Deno with Node and see how Deno overcomes Node's shortcomings.

Deno comparison with Node

Deno was created to address the shortcomings of Node. So, it would be evident that both of them will have many similarities and differences. Let us check each of these.

Feature	Node	Deno
Built using	C++	Rust
JavaScript Engine	V8 (C++ bindings and backend)	V8 (Rust bindings and backend)
Event loop	libuv	tokio
TypeScript support	Need to be installed	Out of the box
Package manager	Npm (centralized)	no package manager (decentralized)
Import system	CommonJS (require())	ES modules (import)
Security	Full access by default	No access by default
package.json	Used	Not used

Deno features

Deno has excellent support for modern features and includes many features not available in Node out of the box (TypeScript support). Let us have a look at some of its main features.

- External dependencies are loaded using URLs
- Code execution in Sandbox

    - Runtime has no access to network file systems and environments
    - Access needs to be granted explicitly
- Typescript support out of the box
- Top-level await
- Standard modules library
- Built-in tools provided like:
    - **Bundler**: Builds scripts and dependencies into a single file
    - **Dependency inspector**: Inspects ES modules and lists all their dependencies
    - **Documents generator:** Parses JSDoc
    - **Formatter**: For auto-formatting JS and TS Files
    - **Test-runner**: For testing using the assertions module from the standard library
    - **Linter**: To catch code issues

Deno foundation stack

Deno has been created using some solid technologies, which increase its chances of becoming highly successful. Once we have gone through the stack used for creating Deno, we can be assured about Deno’s future (a very bright one). Let us check the tech stack used by Deno.

- Uses Google’s V8 Engine
- Written in Rust
- Tokio (async runtime written in Rust) used for event thread management

The V8 Engine by Google

An open-source JavaScript engine built using C++ by Google. It converts JavaScript code (passed by Deno) into bytecode/machine code. Deno supports Typescript out of the box, so we can also provide Typescript code which gets compiled to JavaScript by Deno. It executes the JavaScript code and passes Deno API-related code (Rust code) to the Rust backend via "*rusty_v8*" bindings written in Rust for the V8 engine by the Deno team.

The combination of an interpreter (Ignition) and compiler (TurboFan) makes V8 produce optimized machine code which runs really fast.

Rust

Rust has been the "*most loved*" language in every StackOverflow survey since 2016 (five years in a row). It is a systems programming language that enables control over low-level details. It provides memory-safety, thread-safety, and type systems. For these and many other features, Rust remains the most loved and wanted language for the last five years in StackOverflow surveys.

- Memory safe by default
- Ownership and borrowing support
- Performance is very close to other low-level languages (C, C++)
- Either one mutable or many immutable references
- No NULL pointers, only options

Used by:

- Amazon—firecracker
- Google—Fuchsia
- Facebook—Mononoke
- Microsoft—IoT Edge

The Deno backend contains a file system, network and environment access and is written in Rust. The asynchronous i/o bindings are handled by tokio library.

Tokio I/O library

It is used to write asynchronous applications with Rust. It is an event-driven, non-blocking i/o platform. It uses thread pools and workers to perform the operations in the background.

Whenever a code related to i/o, network, and so on is passed to the V8 engine, it forwards it to Rust backend using "*rusty_v8*" bindings. The execution of the received code is done by tokio using workers.

- Asynchronous I/O ops using non-blocking event
- Uses Mio as an event queue
- The event loop mechanism is used to monitor the call stack and callback queue
- Thread pool and workers are used to perform the tasks in the background

What is a JavaScript engine

JavaScript engine is used to convert JavaScript/Typescript code into bytecode (interpreter) or optimized machine code (compiler) so the computer can execute it.

There are many JavaScript engines released since release.

- V8 Engine by Google
- SpiderMonkey by Mozilla
- Chakra by Microsoft

In the JavaScript engine, the code is parsed into tokens, which are used to generate the Abstract Syntax Tree. The interpreter uses this tree to generate Byte Code. A profiler monitors the byte code and passes it to the compiler if there is a scope for optimization. Compiler outputs optimized machine code which performs faster than the byte code.

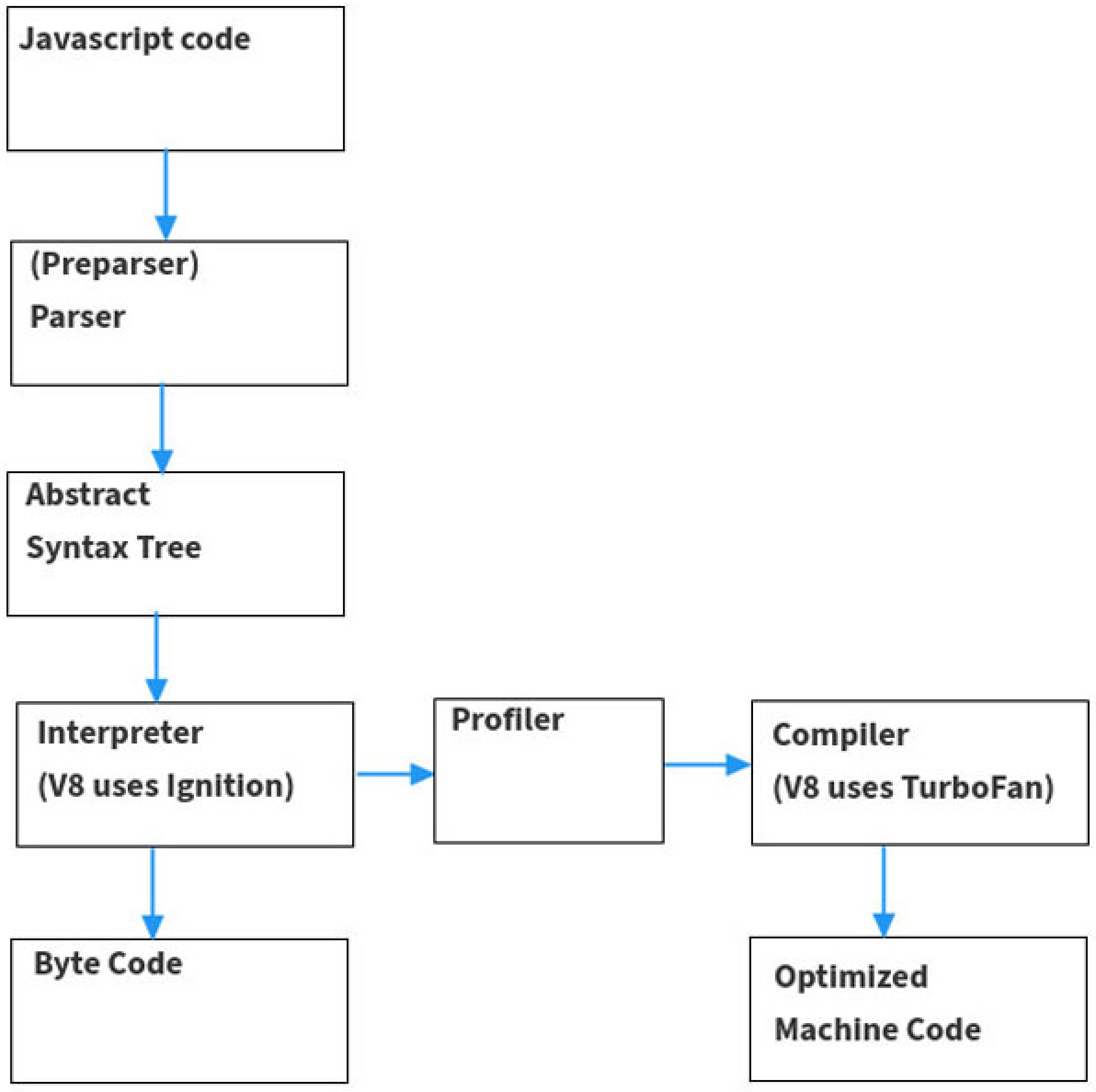
Interpreter:

An interpreter translates the code one line at a time. It provides a ByteCode as output. The interpreter is very fast to get started coding but does not provide any optimizations.

Compiler:

The compiler translates the entire code at once. The compiler has a scope to analyze the entire program and optimize it. It outputs optimized machine code. The compiler takes longer to analyze and optimize the code as it goes through the entire program.

JavaScript engine combines the functionality of interpreter and compiler. This way, we get an optimized code that runs faster. Such a compiler is called the “*JIT Compiler*”. A Profiler is used to monitor the byte code and pass it to the compiler (TurboFan) if there is any scope for optimization.



Javascript Engine Architecture

Figure 1.1: *V8 JavaScript engine*

What is a JavaScript runtime

JavaScript runtime is an environment that combines the JavaScript Engine (V8 in Deno runtime), The API backend written in some low-level language (Rust in Deno runtime), a callback queue and an event loop that continuously monitors the call stack and the callback queue.

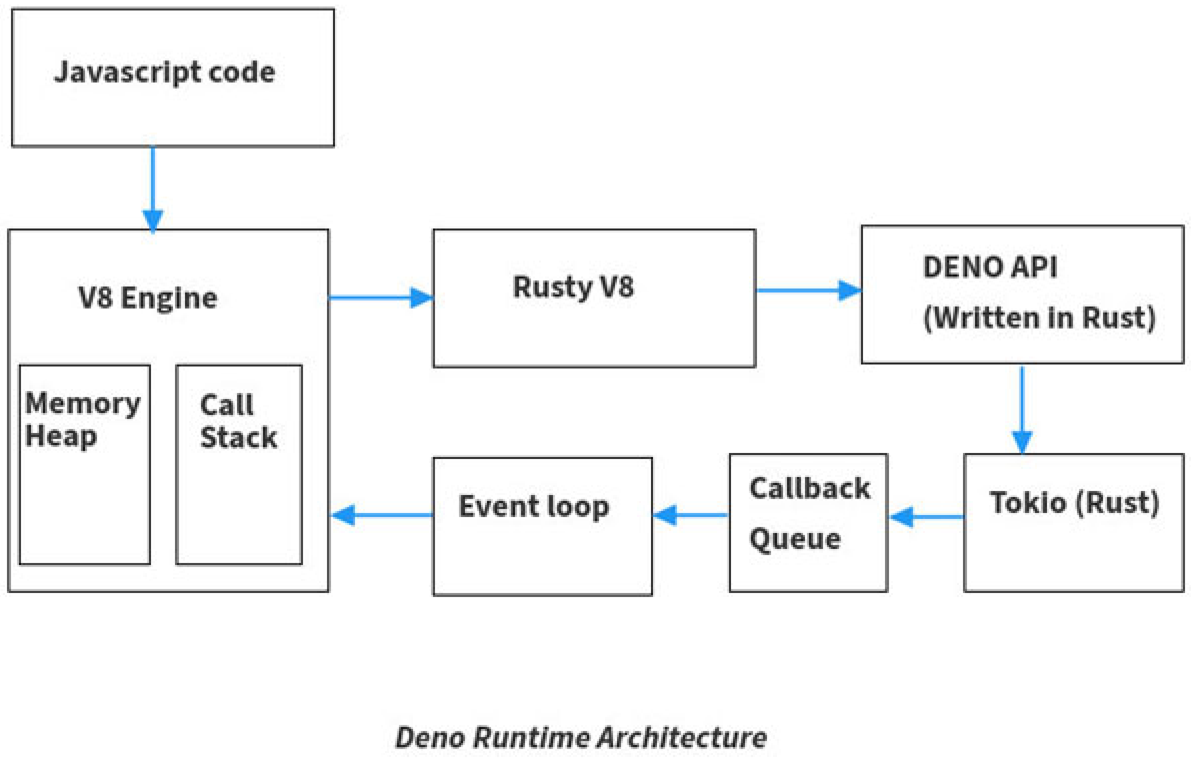


Figure 1.2: *Deno architecture*

Memory heap:

JavaScript code is passed to the V8 engine that stores the variables and functions from the code in the Memory Heap.

Call Stack:

It stores the functions invoked within the code. It operates on the **Last In First Out** (**LIFO**) principle. It always points to the currently running function. Once the function returns or is passed to Deno API, it pops the function out and executes the earlier one on the stack.

Deno API

Whenever Deno detects code related to i/o, network, environment, and so on, it passes the code to the API backend, which is written in Rust. It uses "*rusty_v8*" bindings to pass the code to the backend.

Tokio:

It is an event-driven, asynchronous i/o platform. Deno API passes the code to tokio.

This uses thread pool and workers to execute the operations in the background and passes them on the callback queue.

Callback Queue:

It operates on the **First In First Out** (**FIFO**) principle. It stores all the completed functions passed by Tokio. It "*waits*" till the call stack is empty.

Event loop:

It monitors the call stack and the callback queue continuously. If the call stack is empty and the callback queue has a function in the queue, it passes the function on the call stack where it gets executed.

Conclusion

Our journey in the Deno land has begun with this chapter. We learnt the history of Deno and Node, compared Deno with Node and listed Deno features. We also covered the stack Deno is built from, i.e., *V8* JavaScript engine, *Tokio* event-loop, and Deno API backend written in Rust. Finally, we covered the JavaScript engine and the JavaScript runtime architectures, which are not mandatory but give us a solid foundation to understand what is Deno and how it performs the operations in the background.

While this chapter covered the Major Deno features, TypeScript is one important section in Deno land that needs to be covered. TypeScript support out of the box is a major Deno feature, and learning TypeScript requires an entire dedicated chapter to cover the basics. Hence, we will learn *TypeScript* in the upcoming chapter, as we will be using *TypeScript* for programming throughout this book.

As you have reached this point, it is evident that you are interested and excited to learn Deno. Deno has many exciting and modern features, which we will learn throughout this book.

CHAPTER 2
Introduction to TypeScript

Introduction

TypeScript support out of the box is an exciting feature offered by Deno runtime. TypeScript has been a popular choice for large-scale applications and is being adopted rapidly in the industry. It ranked second in the Most Loved Languages (**https://insights.stackoverflow.com/survey/2020#most-loved-dreaded-and-wanted**) section of the Stack Overflow Developer Survey 2020 (**https://insights.stackoverflow.com/survey/2020**).

Structure

In this chapter, we will cover the following topics:

- Introducing TypeScript
- TypeScript benefits
- TypeScript features
- TypeScript basics
    - Basic types
    - Interfaces
    - Type aliases
    - Interfaces versus Type aliases
    - Type inferences
    - Type assertions
    - Unions

    - Intersections
    - Classes
- Dynamic versus static typing
- Strongly versus weakly typing
- TypeScript setup

Objectives

This chapter should help you become familiar with TypeScript fundamentals and the use of TypeScript for programming Deno applications. This chapter should also help you understand the pros and cons of static type checking and the benefits provided by TypeScript.

Introducing TypeScript

Announced publicly by *Anders Hejlsberg* in 2012, TypeScript is a very popular and open-source superset of JavaScript. It adds static type definitions to JavaScript code. It was developed and maintained by Microsoft. TypeScript provides type safety, type inference, type assertions, and many more features, including ES Next features to JavaScript. Please take a look at the following screenshot:

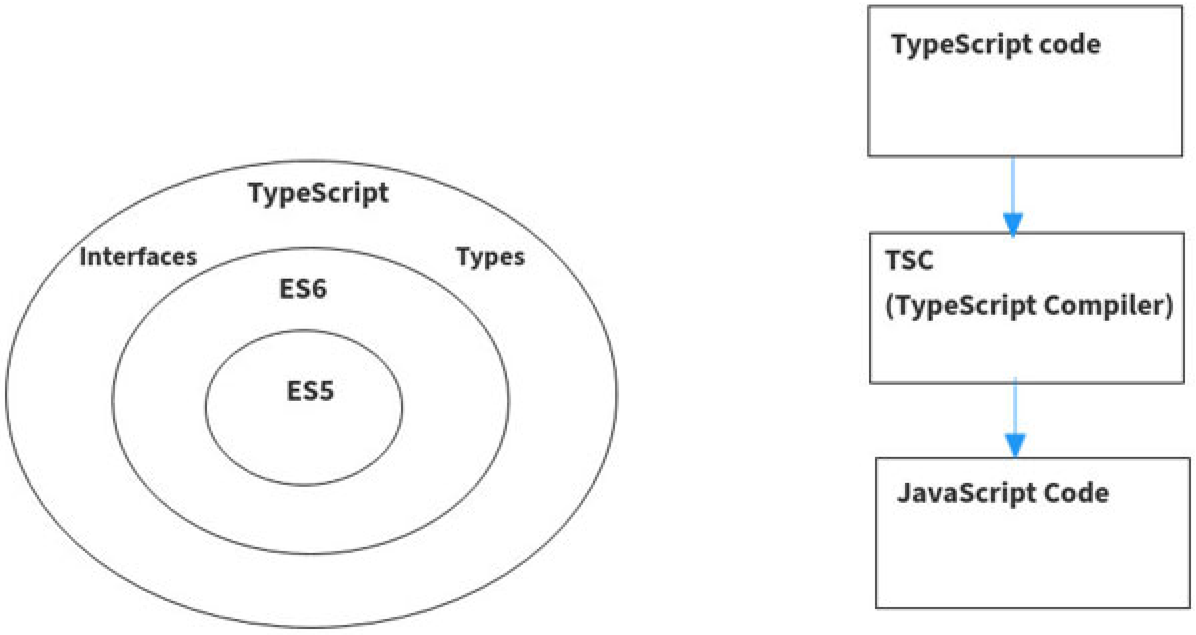


Figure 2.1: *TypeScript*

TypeScript benefits

JavaScript is very popular and widely used in our industry. But while using JavaScript for large-scale applications, its dynamic nature lets many type-related errors and bugs pass through in the development phase and are caught only during runtime. Runtime debugging is more awful compared to compile-time debugging.

Statically typed languages caught errors and bugs during compilation, even before running the code. TypeScript helps us write and maintain large-scale applications by providing type safety during the development cycle. TypeScript catches type-related or other errors during compile time, saving a lot of our precious time.

TypeScript was introduced to add types to JavaScript. TypeScript code gets transformed into Javascript by TypeScript compiler or babel transpiler. TypeScript enables *static type checking*, which detects errors in the code without running it, that is, in compile time. It preserves the runtime behavior of JavaScript, meaning the code will run the same way. Keeping the same runtime behavior is one of the core features of TypeScript.

TypeScript features

TypeScript provides many es-next features out of the box, along with features helpful for developing large-scale applications. We will list some of the major features of TypeScript.

- Detects errors in compile-time, without running the code
- IDE can use type definitions for auto-completion
- Static type system support for JavaScript code
- Less prone to errors due to static type checking
- ES-Next Features support out of the box
- Tooling
- Type safety
- Type inference
- Type assertions
- Generics
- Interfaces and type aliases
- Language services
- Same runtime behavior as that of JavaScript

Basics of TypeScript

Being a superset of JavaScript, TypeScript is easy to grasp for a JavaScript developer. Assuming you are familiar with JavaScript, we will learn the additional types included in TypeScript.

Basic types

We are covering the commonly used data types in this section. We will be using them for the majority of the time during Deno’s development.

- **Boolean**: Accepts either *true* or *false* value.
- **Number**: Accepts either floating point values (for example, 10.34) or integer values (for example, 39).
- **String:** Accepts texts and characters, including *template string.*
- **Array:** Array of mentioned element type declared in two ways:
    - [] symbol:

      ```
      const shoppingList: string[] = [“milk”, “vegetables”];
      ```

    - Generics:

      ```
      const shoppingList: Array<string> = [“milk”, “vegetables”];
      ```

- **Tuple**: An array with a fixed number of elements of either similar or different types. Tuple can be defined as:

  ```
  const deviceStage: [number, string] = [0, “OFF”];
  ```

- **Enum:** Maps string values to a set of numeric values.

  ```
  enum State {
  OFF,
  ON,
  }
  console.log(State.OFF, State[0]); // 0, OFF
  ```

- **Unknown:** Accepts all types. This type of value is only assignable to either **unknown** or **any** type.

- **Any:** It also accepts all types of values like unknown, but it is more permissible than an unknown type. It lets us skip type checking for the variable.
- **Null:** Accepts null value.
- **Undefined:** Accepts undefined value.
- **Never:** Used as a return type of function that never returns.
- **Void:** Used as a return type of function that does not return any value.
- **Object:** Represents non-primitive types.

Interfaces

Interfaces are used to define the shape of an entity or an object. It describes the properties an object should have.

```
interface IPerson {
  firstName: string;
  lastName: string;
}

const person: IPerson = {
  firstName: “first”,
  lastName: “last”
};
```

In the preceding example, **`IPerson`** is an interface (notice the capital I for interface) that defines a shape that the object **`person`** must have. Hence, the property **`place`** will cause a compiler error as it does not exist in the interface **`IPerson`**.

Optional properties

In many cases, we need to provide a field that can be optional. In an interface, “?” symbol can be used to make the field optional.

```
interface IPerson {
  name: string;
  age?: number; // age is optional
}

let printProfile = (person: IPerson): void => {
  console.log(person.name);
};

const person = { name: “name1” };
printProfile(person);
```

Read only properties

A **readonly** keyword before the property indicates we want a property to be immutable after creation.

```
interface IPerson {
  name: string;
  readonly age: number; // age is read-only
}

const person: IPerson = {
  name: “me”,
  age: 29,
};

person.name = “you”;
```

```
// person.age = 25; // Cannot assign to age because it is a read-only property
```

Type aliases

Type aliases work in a similar way to that of interfaces. While interfaces are used to define the shape of an object, type aliases can be used to provide names for primitive types. They can also be used for union types.

```
type myInt = number; // type provides new name myInt for primitive type number
let myId: myInt = 11; // Here myInt represents number type
let id: number = 11;

// Unions
type unionId = number | string; // union types
let myUnionId: unionId = 25; // number
myUnionId = “1a2b”; // string

// Similar functioning to interfaces
type PersonType = {
  firstName: string;
  lastName: string;
};

const person: PersonType = {
  firstName: “first”,
  lastName: “last”,
   // place: “unknown”, // Compiler error: place does not exist in the type IPerson
};
```

Interfaces versus type aliases

Interface merging is allowed in TypeScript, whereas type aliases merging will give a compile error.

```
interface IPerson {
  name: string;
}

interface Iperson {
  // Merges place into existing Iperson
  place: string;
}

const person: Iperson = { name: “Name1”, place: “space” };

type PersonType = {
  name: string;
};

// type PersonType = {
//   // Compiler error: Duplicate identifier ‘PersonType’
//   place: string;
// };

// const person1: PersonType = { name: “name2”, place: “time” }; // place does not exist
```

Type inference

There are two ways to handle type inference in TypeScript. When we provide the type *explicitly*, it is called **Explicit**

typing, and when TypeScript infers the type *implicitly* using the assigned value, it is called **Implicit typing**.

- **Explicit typing**

  Whenever we provide the type to a variable explicitly, it is called explicit typing.

  ```
  let myId: number = 25;   // Explicitly typed as number
  ```

- **Implicit typing**

  TypeScript infers the data type of a variable even if the type is not specified explicitly either by the initially assigned value or function parameter default value, or function return type. This is called implicit typing or type inference.

  If **`false`** is assigned to **`isAllowed`** variable without explicit annotation, TypeScript concludes the data type to be **`boolean`** using the **`false`** value.

  ```
  let isAllowed = false; // Implicitly typed
  // TypeScript infers type boolean as the value is false
  // let isAllowed: boolean = false;

  // Compiler error
  // isAllowed = 25; // TS2322 [ERROR]: Type ‘number’ is not assignable to type ‘boolean’.
  ```

If no value is provided during the declaration, **`any`** type is inferred. This behavior can be disabled by setting **`“noImplicitAny”: true;`** flag in **`tsconfig.json`**.

```
let anyVar; // ‘any’ type inferred
anyVar = 2.3;
anyVar = false;
anyVar = “I’m any”;
```

Type assertions

Type assertions are used to override types inferred by the compiler. Type assertions can be defined in two ways:

- **`as`** syntax

```
const word: unknown = “abcdefg”;
// const charCount: number = word.length; // error: TS2571 [ERROR]: Object is of type ‘unknown’.
const charCount: number = (word as string).length;
```

- Angle bracket <> syntax

```
const word: unknown = “abcdefg”;
// const charCount: number = word.length; // error: TS2571 [ERROR]: Object is of type ‘unknown’.
const charCount: number = (<string>word).length;
```

Unions

In union types, a variable can hold values of two or more types. Union is denoted by the vertical bar or pipe “|” symbol.

```
let myNum: number = 24;
// myNum = “id2234”; // error: TS2322 [ERROR]: Type ‘string’ is not assignable to type ‘number’.
let myId: string | number = 25; // myId will accept either string or number.
myId = “id2234”; // string is accepted

// Using type
type myIdType = string | number;
let myId1: myIdType = 53;
myId1 = “id1234”;
```

Intersections

Intersection types are created by combining all the properties and methods of two different types. Ampersand is used to create intersection types.

```
interface IPerson {
  name: string;
  age: number;
}

interface IEmployee {
  employeeId: number;
}

// PersonType will have all the props of IPerson & IEmployee
type PersonType = IPerson & IEmployee;

const printMe = (person: PersonType): PersonType => {
  return person;
};

const user: PersonType = {
  name: “name1”,
  age: 25,
  employeeId: 555,
};

const result = printMe(user);
console.log(result);
```

Classes

TypeScript allows an object-oriented programming approach with the help of classes. In **OOP** terms, class is a blueprint used to create an instance or object. In general, a class can have the following entities:

- **`constructor()`**: Invoked whenever an instance of the class is to be created.
- **fields:** A variable or property.
- **method:** A function

Modifiers

Modifiers are access controllers for the fields of the class. There are three modifiers named **`public`**, **`private`**, and **`protected`**, used for fields and methods.

public

This is the default modifier which allows access to the field/method outside class.

```
class Person {
  name: string; // public
  age: number; // public

  // Called during instance creation
  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }

  printMe() {
    console.log(this.name, this.age); // this refers to ‘person’ instance
  }
}

const person = new Person(“me”, 99); // Calls contructor,with name=”me” and age=99
person.printMe();
console.log(person.age); // public property is accessible outside class
```

private

It does not allow access to the field/method outside class. Even the subclasses do not have access to private properties. It is defined by either a private keyword or ECMAScript’s hash(#) modifier.

```
class Person {
  name: string;
  private _age: number;
  #isActive: boolean; // private identifier

  constructor(name: string, age: number, isActive: boolean = true) {
    this.name = name;
    this.age = age;
    this.#isActive = isActive;
  }

  printMe() {
    console.log(this.name, this.age, this.#isActive);
  }
}

const person = new Person(“me”, 99);
person.printMe();
console.log(person.name);

/**
* error: TS2341 [ERROR]: Property ‘age’ is private and only accessible within class ‘Person’.
*/
// console.log(person.age);

/**
* TS18013 [ERROR]: Property ‘#isActive’ is not accessible
* outside class ‘Person’ because it has a private identifier.
*/
// console.log(person.#isActive);
```

protected

It does not allow access to the field/method outside class, but the subclasses do have access to its fields.

```
class Person {
  name: string;
  protected age: number; // protected field

  constructor(name: string = “unnamed”, age: number = 0) {
    this.name = name;
    this.age = age;
  }

  printMe() {
    console.log(this.name, this.age);
  }
}

class User extends Person {
  userId: number;

  constructor(userId: number, name: string, age: number) {
    super(name, age);
    this.userId = userId;
  }

  printMe() {
    console.log(this.name);
       console.log(this.age);  //  protected  field  age  of
Person class is accessible to subclasses
  }
}

const user = new User(1, “me”, 99);
user.printMe();

/**
* error: TS2445 [ERROR]: Property ‘age’ is protected and
only accessible within
```

```
* class ‘Person’ and its subclasses.
*/
// console.log(user.age);
```

Getter and Setter

As we saw in the modifier section, **`private`** and **`protected`**, fields/methods are not accessible outside the class. We will need to access them in some situations. *Getters* and *Setters* will be helpful in such cases.

```
class Person {
  name: string;
  private _age: number;
  #isActive: boolean;

  constructor(name: string, age: number, isActive: boolean = true) {
    this.name = name;
    this.age = age;
    this.#isActive = isActive;
  }

  printMe() {
    console.log(this.name, this.age, this.#isActive);
  }

  get _age() {
    return this.age;
  }

  set _age(value: number) {
    this.age = value;
  }

  get Is_active() {
    return this.#isActive;
  }
  set Is_active(value: boolean) {
    this.#isActive = value;
  }
}

const person = new Person(“me”, 99);
person.printMe();
console.log(person.name);

/**
* error: TS2341 [ERROR]: Property ‘age’ is private and only accessible within class ‘Person’.
*/
// console.log(person.age);

console.log(person._age);
person._age = 11;
console.log(person._age);

/**
* TS18013 [ERROR]: Property ‘#isActive’ is not accessible
* outside class ‘Person’ because it has a private identifier.
*/
// console.log(person.#isActive);

console.log(person.Is_active);
person.Is_active = false;
console.log(person.Is_active);
```

Static

It is used to define a property that belongs to the class and not its instance. This means that the static field remains the same for all class instances. It is useful in scenarios where we want a single copy of a field shared across all the instances. In the following example, we are using **`static instanceCount`** to count the number of instances of the **`Person`** object.

```
class Person {
  name: string; // public
  age: number; // public

  // static property shared across instances
  static instanceCount: number = 0;

  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;

    // TS2576 [ERROR]: Property ‘instanceCount’ is a static member of type ‘Person’.
    // this.instanceCount += 1;

    // Class property
    Person.instanceCount += 1;
  }

  printMe() {
    console.log(this.name, this.age);
  }
  printTotalPersons() {
    console.log("Total persons: ", Person.instanceCount);
  }
}

const person = new Person("me", 99);
person.printMe();
person.printTotalPersons(); // instanceCount = 1;

const person2 = new Person("you", 29);
person2.printMe();
person2.printTotalPersons(); // instanceCount = 2;
```

Abstract

Abstract class is a common base class (for example, **`Person`**) with a basic structure that can be inherited by other classes (for example, **`Employee`**, **`Employer`**).

```
abstract class Person {
  name: string; // public
  age: number; // public

  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }

  printMe() {
    console.log(this.name, this.age);
  }
}

/**
* error: TS2511 [ERROR]: Cannot create an instance of an abstract class.
*/
// const person = new Person(“employee”, 39);
// person.printMe();

class Employee extends Person {}
class Employer extends Person {}

const employee = new Employee(“employee”, 39);
employee.printMe();

const employer = new Employer(“employer”, 99);
employer.printMe();
```

Static versus dynamic typing

While writing applications, we come across programming languages which fall into either static or dynamic type systems. Static typing provides type safety during compile time but takes more time in the development process, whereas dynamic typing allows us to write code faster but may have errors and bugs which get caught only during run time.

```
// Static typing (TypeScript)
let id: number = 235; // Type is determined in compile-time
// id = “332w”; // Error. Type change not allowed
// Type inferred by TypeScript
let myId = 235; // number type inferred by value
// myId = “332w”; // Error. Type change not allowed

// Dynamic typing (JavaScript)
let id1 = 10; // Type is determined in the runtime
id1 = “132w”; // Type change allowed
```

	Static typing	**Dynamic typing**
Data type	Needs to be specified explicitly	No need to specify data type explicitly
Type checking	At compile-time	At runtime
Type change	Not allowed	Allowed
Languages	C, C++, Java, C#	Python, JavaScript, Ruby

Strong versus weak typing

Strong typing means type conversions are not allowed implicitly, whereas weak typing means type conversions can be done implicitly.

```
# Strong typing (Python)
```

1. `id = 34 + “sfs” // Error: unsupported operand type(s) for +: ‘int’ and ‘str’`

```
// Weak typing (JavaScript)
```

1. `let id = 34 + “abc”; // results “34abc” because of type casting`

	Strong typing	**Weak typing**
Languages	Python	JavaScript
Type conversion	Only manual	Automatic and manual
Example	1 + “2” // gives error in Python	1 + “2” // results “12” in JavaScript

TypeScript setup

Before Deno, TypeScript had to be installed via “**`npm`**”. But, as Deno ships with TypeScript, we can start writing TypeScript code directly. But, if you intend to use TypeScript outside the Deno environment, you will have to install it using npm globally.

Note: Make sure you have installed node & npm.

```
npm install -g typescript
```

Once done, check if typescript is installed by running:

```
tsc --version
tsc --help
```

tsconfig.json

It is a TypeScript configuration/settings file that indicates the root of the TypeScript project and also specifies root files and compiler options. Deno provides a default **`tsconfig.json`**, which can be checked here.

(**https://deno.land/manual/getting_started/typescript#custom-typescript-compiler-options**)

Conclusion

TypeScript is a widely revered and accepted superset of JavaScript providing type safety. Its popularity in the industry is one of the reasons Deno made it the primary language in its setup.

We have covered the TypeScript basics in this chapter, and there is much more we can learn in the entire journey, as TypeScript will be our language of choice for Deno programming. We will get hands-on practice with TypeScript in upcoming chapters.

Meanwhile, you can always try different TypeScript features directly in the TS Playground (**https://www.typescriptlang.org/play/**).

With this chapter, our foundation section (Deno, TypeScript) is completed, and we will get started with hands-on programming with Deno in the upcoming chapter, where we will setup Deno and get to know basic Deno commands.

CHAPTER 3
Setting Up Deno

Introduction

We are now ready to get our hands-on experience running Deno in this section. Before getting started, we need to set up the development environment for Deno. Coding or programming is only one of the major aspects of software engineering. There are other major factors involved that allow us to engineer and ship well-crafted software applications in an estimated timeframe.

Structure

In this chapter, we will cover the following points:

- Installing Deno
- Development tools
    - Terminal
    - zsh + Oh my zsh
    - Vim
    - VS Code (with extensions)
    - Git and GitHub
    - Postman
    - Docker
    - Silver searcher
    - Database
        - MongoDB + MongoDB Atlas
        - Postgres

        - Redis
    - Heroku CLI
    - Pomodoro Technique
    - Tmux

Objectives

We will be installing Deno via terminal, and we will also have a brief overview of some major development tools and technologies required during Deno development. We will be using them while working on the projects later on. Though optional, these tools will provide a better developer experience and organization. We will also learn to set up some of the popular development tools useful for developers to be productive and organized.

Installing Deno

Installing Deno is quite easy and can be done using a single terminal command. There are various ways to install Deno on your machine as follows:

```
# Shell (MacOS, Linux):
curl -fsSL https://deno.land/x/install/install.sh | sh
Install specific version:
curl -fsSL https://deno.land/x/install/install.sh | sh -s
v1.5.4
# PowerShell (Windows):
iwr https://deno.land/x/install/install.ps1 -useb | iex
Install specific version:
$v=”1.5.4”; iwr https://deno.land/x/install/install.ps1 -useb
| iex
# Homebrew (MacOS):
brew install deno
```

Check if Deno is installed by running:

```
deno --version
```

Deno can be upgraded by running:

```
deno upgrade
```

Deno can be upgraded to a specific version by running:

```
deno upgrade --version 1.5.3
```

Development tools

Productivity, focus, and organization are critical aspects of a developer’s day-to-day life. They allow us to manage and organize our workflow so we can create more value with a systematic and organized approach.

Development tools are one important aspect of software engineering that is often ignored or sidelined. These tools are very helpful for increased productivity and better organization. They let us focus on the prime part, that is, software development, whereas they manage other factors efficiently for us. Let us have an overview of some of the major tools used by developers worldwide.

Terminal

This is where we live in the programming world for most of our time. Getting familiar with it gives us an upper edge while planning and executing daily tasks or scripts as they are known in the software world. This is the place where we will execute Deno commands (from installing Deno to running Deno applications). You could use UNIX terminal for Linux/mac or PowerShell for windows OS.

zsh + oh my zsh

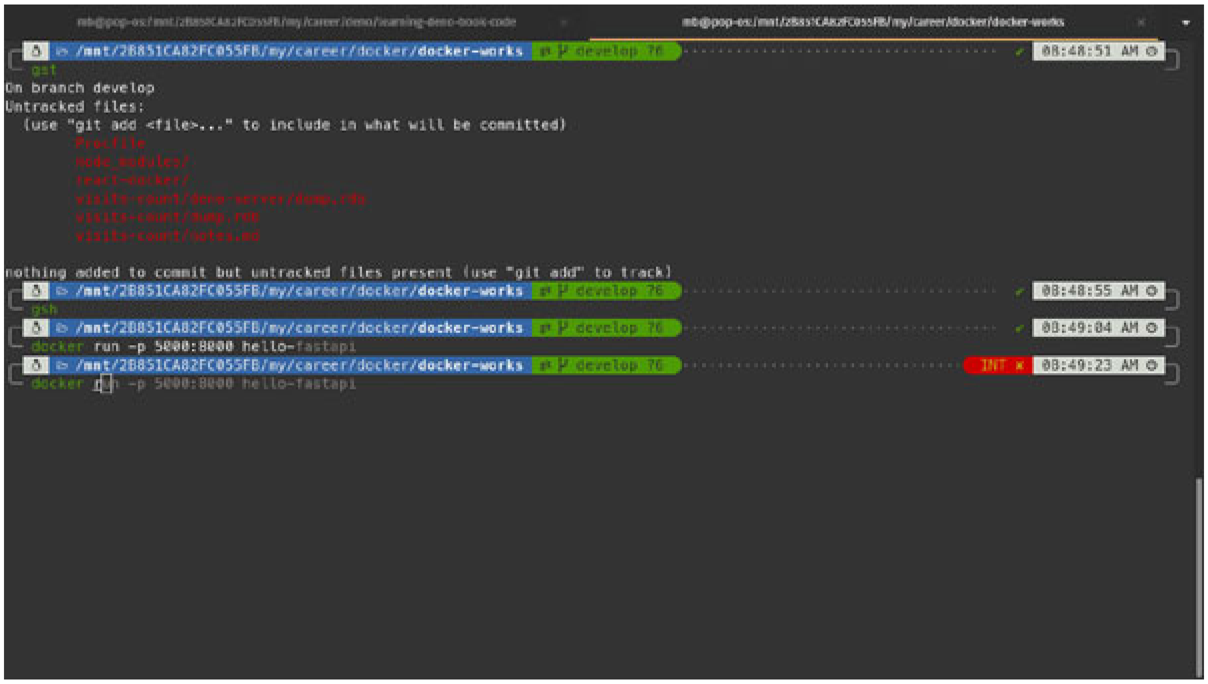
While using the terminal, **`zsh`** (Z shell) and **`oh my zsh`** (zsh framework) can level up your overall development experience and productivity. Check out the GitHub page for installation guides. Oh my zsh comes with some great plugins, as shown in *figure 3.1*.

`Install zsh`

https://github.com/ohmyzsh/ohmyzsh/wiki/Installing-ZSH

`Install oh my zsh`

https://github.com/ohmyzsh/ohmyzsh



```
/mnt/2B851CA82FC055FB/my/career/docker/docker-works  develop 76 ........ ✓ 08:48:51 AM
gst
On branch develop
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        Procfile
        node_modules/
        react-docker/
        visits-count/deno-server/dump.rdb
        visits-count/dump.rdb
        visits-count/notes.md

nothing added to commit but untracked files present (use "git add" to track)
/mnt/2B851CA82FC055FB/my/career/docker/docker-works  develop 76 ........ ✓ 08:48:55 AM
gsh
/mnt/2B851CA82FC055FB/my/career/docker/docker-works  develop 76 ........ ✓ 08:49:04 AM
docker run -p 5000:8000 hello-fastapi
/mnt/2B851CA82FC055FB/my/career/docker/docker-works  develop 76 .... INT ✘ 08:49:23 AM
docker run -p 5000:8000 hello-fastapi
```

Figure 3.1: *Terminal with zsh + oh my zsh setup*

Theme: powerlevel10k

Plugins: Git, zsh-syntax-highlighting, and zsh-autosuggestions

[Vim]()

Vim (also known as vi) is a legendary text editor used for fast and efficient text operations. Vim has been in the industry for decades and has maintained its ranking in the topmost editors for all these years. It is a keyboard-centric application that focuses on keyboard commands to perform complex text operations. Vim has its own keybinding, which needs some time to get used to, resulting in better control over text manipulation and cursor movements.

There are many legends associated with vim, and editor war is perhaps the most famous of all. There is a famous legend about the rivalry between vim and emacs back in the 90s. Vim can be used via VS Code Extension or directly from the terminal.

```
Install vim
```

https://www.vim.org/download.php

VS Code

VS Code is an open-source IDE developed by Microsoft. It is available on all platforms (Windows, Linux, and macOS) and is one of the most popular and widely used IDE at the moment.

It is built with Electron—A JavaScript framework for the development of cross-platform desktop applications. We will be using it for coding Deno applications. Installing VS Code on any platform is very easy. Open **https://code.visualstudio.com/download** and download and run the executable file for your OS.

VS Code extensions

We will list a few useful extensions while developing Deno programs in VS Code. These extensions can be searched and installed from VS Code.

- **Deno**: Enables Deno support for VS Code
- **Prettier**: A code formatter to prettify your Deno code
- **ESLint**: A linter that analyses your ES code for issues
- **Vim**: Enable vim support in VS Code
- **GitLens**: Visualized git experience in VS Code

Git and GitHub

Git is a distributed version control system developed by *Linus Torvalds* (Creator of the Linux kernel). It is widely used in the Software Industry for code versioning and management. Head over to **https://git-scm.com/downloads** to install it on your OS.

GitHub is a source code hosting site owned by *Microsoft*. It uses Git for version control and source code management. Head over to **https://github.com/** and sign up if you have not yet. We will use it for hosting our Deno code.

Postman

A platform for API development and testing, it helps us manage the API collections and test our APIs with GUI. We will be using it to test our APIs. Head over to **https://www.postman.com/downloads/** to download and install Postman on your OS platform.

Docker

Docker is a container-based software development platform that runs the entire application as an isolated process. It is a very useful tool in containerization and microservices. Docker has brought the container revolution that changed the way an application is developed and deployed.

Docker was ranked #1 most wanted, #2 most loved, and #3 most popular platforms in Stack Overflow survey 2020.

`Install docker`

https://docs.docker.com/engine/install/

Docker Compose

Docker Compose is a tool developed by the Docker team to compose and run multiple docker containers as a single service. We will be using it to define multiple services/containers in the **`docker-compose.yml`** file. We can then run all the defined services with the YAML configurations using a single command.

`Install docker-compose`

https://docs.docker.com/compose/install/

Silver Searcher

A free and open-source command-line tool for searching code fast from the terminal. Check out **https://github.com/ggreer/the_silver_searcher** for installation instructions. This tool will be helpful for quick code searches from the terminal during Deno development.

Databases

There are two types of databases widely used in the industry, *relational* databases (SQL and PostgreSQL) storing data in tables as columns and rows and *non-relational* databases (MongoDB, Redis, and ElasticSearch) storing the data in collections. Postgres (RDBMS) and MongoDB (NoSQL) databases are the most popular amongst databases.

PostgreSQL

PostgreSQL (also known as **postgres**) is a free and open-source **relational database management system** (**RDBMS**). It is a SQL-based database that uses SQL for data management and stores the data in table format.

`Install postgres`

https://www.postgresql.org/download/

MongoDB and MongoDB Atlas

MongoDB is an open-source NoSQL database system that uses JSON-like documents to store the data. There are two ways to use MongoDB in Deno projects as follows:

1. **Install MongoDB locally**

   This will install MongoDB on your machine, which can be used as a database.

   During the local development of Deno applications.

   https://docs.mongodb.com/manual/installation/

2. **MongoDB Atlas**

   It is a cloud-based database service that lets us use MongoDB services without installing them locally. It has a free-tier plan that we will be using as a database for Deno projects. You can register yourself to get started with MongoDB Atlas.

   https://www.mongodb.com/cloud/atlas

Redis

Remote Dictionary Server (**Redis**) is an open-source, in-memory database that stores data in key-value format. We can always try it locally by following instructions on **https://redis.io/topics/quickstart**

Redis docker image

https://hub.docker.com/_/redis

Heroku

It is a **platform as a service** (**PAAS**) used to deploy applications written using different programming languages for free.

Signup on Heroku

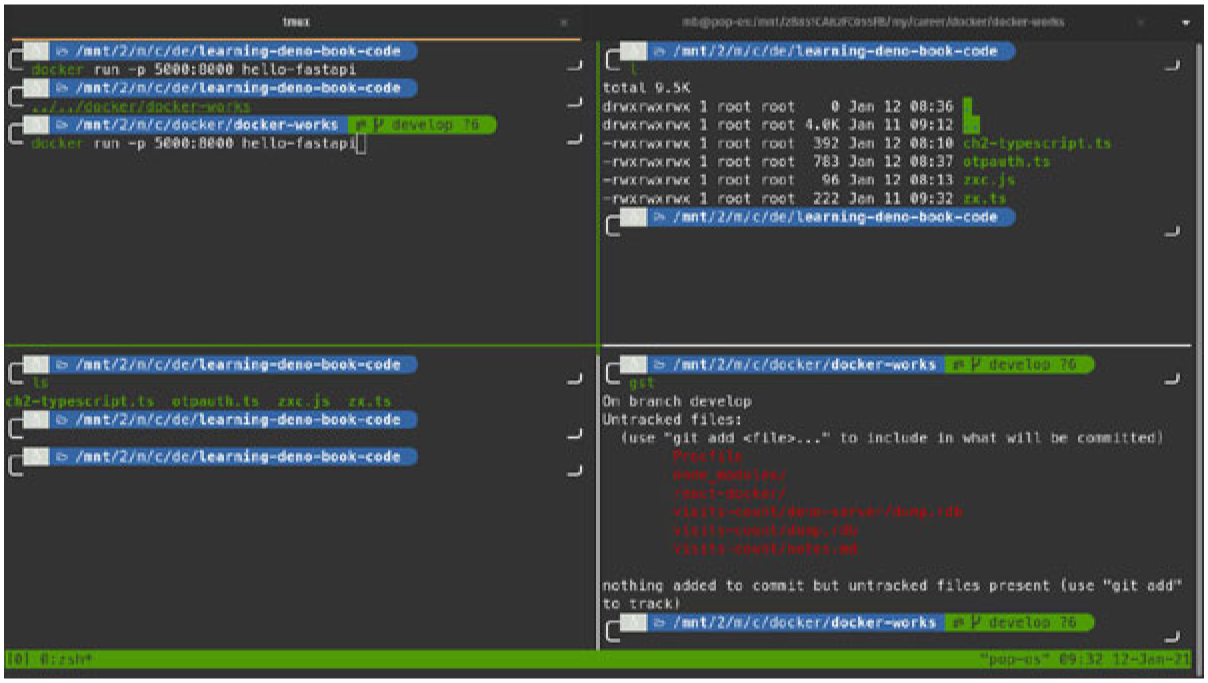
https://www.heroku.com/home

Heroku CLI

https://devcenter.heroku.com/articles/heroku-cli#download-and-install

Tmux

It is an open-source terminal multiplexer program that lets you create multiple terminals from a single screen. It will be a very useful utility while using the terminal. Please take a look at the following screenshot:



```
/mnt/2/m/c/de/learning-deno-book-code
docker run -p 5000:8000 hello-fastapi
/mnt/2/m/c/de/learning-deno-book-code
../../docker/docker-works
/mnt/2/m/c/docker/docker-works  develop ?6
docker run -p 5000:8000 hello-fastapi
```

```
/mnt/2/m/c/de/learning-deno-book-code
l
total 9.5K
drwxrwxrwx 1 root root    0 Jan 12 08:36 .
drwxrwxrwx 1 root root 4.0K Jan 11 09:12 ..
-rwxrwxrwx 1 root root  392 Jan 12 08:10 ch2-typescript.ts
-rwxrwxrwx 1 root root  783 Jan 12 08:37 otpauth.ts
-rwxrwxrwx 1 root root   96 Jan 12 08:13 zxc.js
-rwxrwxrwx 1 root root  222 Jan 11 09:32 zx.ts
/mnt/2/m/c/de/learning-deno-book-code
```

```
/mnt/2/m/c/de/learning-deno-book-code
ls
ch2-typescript.ts  otpauth.ts  zxc.js  zx.ts
/mnt/2/m/c/de/learning-deno-book-code

/mnt/2/m/c/de/learning-deno-book-code
```

```
/mnt/2/m/c/docker/docker-works  develop ?6
gst
On branch develop
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        Procfile
        node_modules/
        react-docker/
        visits-count/deno-server/dump.rdb
        visits-count/dump.rdb
        visits-count/notes.md

nothing added to commit but untracked files present (use "git add"
to track)
/mnt/2/m/c/docker/docker-works  develop ?6
```

```
[0] 0:zsh*                                    "pop-os" 09:32 12-Jan-21
```

Figure 3.2: *Tmux*

Pomodoro Technique

It is a time management technique used to break down work into time intervals known as **Pomodoro** (the Italian word for "*tomato*"), followed by a short/long break. It helps us break our work into smaller tasks and work on each task for certain Pomodoros/Pomodori. It has been beneficial to many developers (including myself), and hence, it has made its way into this chapter. You can learn more about it from the official website:

https://francescocirillo.com/pages/pomodoro-technique

Term	Length
Pomodoro	25 minutes
Short break	5 minutes
Long break	**15 or 25 minutes**

Table 3.1: *Pomodoro technique terms*

Conclusion

We have explored some of the major tools useful during software development. We will be using many of the mentioned tools while creating Deno applications. Though many of these tools are optional for Deno development, we still recommend using them to have a better development experience. We will learn about the Deno Ecosystem, including CLI, Standard modules, and the nest.land (Deno package registry that uses blockchain), and other stuff in the upcoming chapter.

CHAPTER 4
Deno Ecosystem

Introduction

Before getting hands-on experience in Deno applications development, we need to learn the runtime APIs, CLI commands, and other functions. These APIs will provide input/output and network-related functionality. We will explore the Deno CLI, Deno runtime APIs (Web APIs + Deno global APIs), standard modules library, third-party modules, and module registries that together form the Deno ecosystem.

Structure

In this chapter, we will cover the following points:

- Deno CLI
  - Environment variables
  - Subcommands
  - Passing arguments
  - File watcher
  - Permission flags
  - Permissions allow-list
- Deno runtime APIs
  - Web APIs
  - Deno global APIs
  - Variables
  - Classes

- Deno modules
    - Standard library
    - Third-party modules
- Module registries
    - deno.land/x
    - nest.land
    - Skypack CDN
    - esm.sh
    - jspm
    - Web
- Dependency management

Objectives

We will be exploring the various sections of the Deno ecosystem in this chapter. The ecosystem includes **command line interface** (**CLI**) commands such as run, install, and so on. The runtime APIs (Web + Deno global) and standard modules (inspired by Go's standard library) such as uuid, http, fs, and so on. Apart from the standard modules, there are thousands of third-party Deno modules available for various applications such as *http server (oak, opine), graphql (oak-graphql), script runner (velociraptor), orm (denodb), module managers (trex), file watcher (denon),* and so on.

We will be using some of them in the upcoming Deno projects section. These modules are hosted on Deno's official registry at **https://deno.land/x** and also at **https://nest.land**, which uses Permaweb where modules can never be deleted. As Deno can import modules from URL, modules hosted on, esm.sh, and skypack can also be used for Deno application development.

Deno CLI

Deno **command-line interface** (**CLI**) provides some standard environment variables, subcommands, and options out of the box. We will check some major ones in this section.

Environment variables

Environment variables are stored outside the code program and can be modified without any changes to the code itself. These are very useful for setting values in different execution environments (development, staging, production, and so on). They are also used to store private/secret keys that should not be exposed as part of the code. Deno offers several environment variables, and we are listing a few major ones as follows:

- **`DENO_DIR`**: The directory used to cache downloaded imports. Imports are downloaded only on the first execution. The cached imports from DENO_DIR are then used for subsequent runs
- **`DENO_INSTALL_ROOT`**: The directory used to save modules installed via Deno install command.

Subcommands

Deno subcommands are used to perform a variety of operations, such as running the program, formatting the code, installing scripts, and so on. We will be using the **`run`** command most frequently along with the **`help`** to get help about specific subcommands. We will check most of the subcommands with a brief overview and code example in the following section:

- **help**: Useful to print general help information or of the specified subcommand.

1. `deno help # Lists subcommands, options, environment variables`
2. `deno help run # Prints help info for the run command`

- **run**: It executes Deno program provided in any of the following ways:
  - filename:
    1. `deno run welcome.ts # executes local welcome.ts file`
  - url:
    1. `deno run https://deno.land/std@0.79.0/examples/welcome.ts`
  - stdin:
    1. `cat welcome.ts | deno run -`
- **install:** Installs a Deno script as an executable in the directory specified as **`DENO_INSTALL_ROOT`**.
  1. `# Install denon (file watcher) as an executable`
  2. `deno install -qAf https://deno.land/x/denon@2.4.5/denon.ts`
- **fmt:** Formatter for TypeScript/Javascript code.
  1. `deno fmt # Formats all the files in the current directory recursively`
- **doc**: Prints documentation for provided source file (module)
- **eval**: Evaluate Javascript code from the terminal
  1. `deno eval “console.log(Deno.args)” deno land`
  2. `# Output: [“deno”, “land”]`
- **test**: Runs tests using a built-in test runner. Files with **`{*_,*.,}test.{js,mjs,ts,jsx,tsx}`** patterns are considered

for testing. Supports filter, coverage, fail-fast, and other options.

- **cache**: Downloads and compiles a module and saves all its dependencies in the local cache without running the code.

  1. `deno cache https://deno.land/std/http/file_server.ts`

- **bundle**: Bundles a single JavaScript file with all dependencies.
- **completions**: Outputs Deno shell completion script to the provided source file.

  1. `# Install Deno completions as a plugin in oh-my-zsh`
  2. `mkdir ~/.oh-my-zsh/custom/plugins/deno`
  3. `deno completions zsh > ~/.oh-my-zsh/custom/plugins/deno/_deno`
  4. `# Add deno in ~/.zshrc plugin`
  5. `plugins=(… deno) // … represents other plugins`

- **compile**: Compiles provided script into self-contained executable.

Passing arguments

Arguments are useful whenever a dynamic value needs to be provided to the code. Arguments can be passed to the script by specifying them after the script name in the following format:

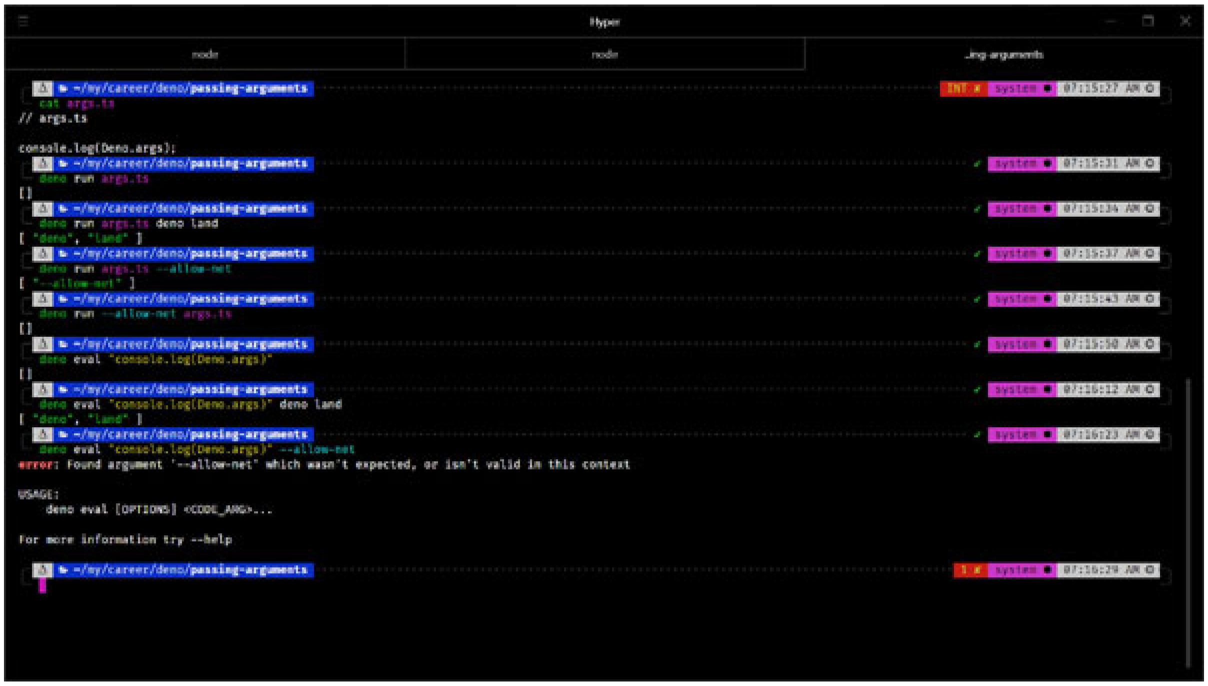
1. `deno run [FLAGS] [SCRIPT] [ARGUMENTS]`

The preceding format starts with **`deno`** executable command followed by **`run`** subcommand, which executes the provided script followed by optional security flags such as **`--allow-net`**, **`--allow-read`**, and so on. The path for the script to be executed is then provided, followed by optional arguments. All the entries (separated by space) provided after the script are considered as arguments and are made available via **`Deno.args`** variable.

We can use the preceding format for running Deno scripts with arguments. Create a file named **`args.ts`** in the projects folder (like **`home/my/work/deno/passing-arguments`**) with the following code:

```
// args.ts
console.log(Deno.args);
```

The preceding code simply logs Deno runtime variable **`Deno.args`**, which is an array of provided arguments. An empty array [] is logged if no arguments are provided, as shown in *figure 4.1*:



```
~/my/career/deno/passing-arguments                    INT ✘  system ●  07:15:27 AM
cat args.ts
// args.ts

console.log(Deno.args);
~/my/career/deno/passing-arguments                        ✓  system ●  07:15:31 AM
deno run args.ts
[]
~/my/career/deno/passing-arguments                        ✓  system ●  07:15:34 AM
deno run args.ts deno land
[ "deno", "land" ]
~/my/career/deno/passing-arguments                        ✓  system ●  07:15:37 AM
deno run args.ts --allow-net
[ "--allow-net" ]
~/my/career/deno/passing-arguments                        ✓  system ●  07:15:43 AM
deno run --allow-net args.ts
[]
~/my/career/deno/passing-arguments                        ✓  system ●  07:15:50 AM
deno eval "console.log(Deno.args)"
[]
~/my/career/deno/passing-arguments                        ✓  system ●  07:16:12 AM
deno eval "console.log(Deno.args)" deno land
[ "deno", "land" ]
~/my/career/deno/passing-arguments                        ✓  system ●  07:16:23 AM
deno eval "console.log(Deno.args)" --allow-net
error: Found argument '--allow-net' which wasn't expected, or isn't valid in this context

USAGE:
    deno eval [OPTIONS] <CODE_ARG>...

For more information try --help

~/my/career/deno/passing-arguments                      1 ✘  system ●  07:16:29 AM
```

Figure 4.1: *Passing arguments*

We will now pass two arguments **`deno`** and **`land`**, to the **`args.ts`** file. As shown in *figure 4.1*, the console should log **`[“deno”, “land”]`** after running the following code:

```
deno run args.ts deno land
# Output: [“deno”, “land”]
```

The preceding code runs the **`args.ts`** file with two additional parameters **`deno`** and **`land`**. As Deno considers all the entries passed after the file name separated with space as arguments, the entries are made available to the **`args.ts`** file in **`Deno.args`** variable.

Note: Flags should not be passed after the script name as they will be considered as arguments.

The **`--allow-net`** flag passed before **`args.ts`** script is considered as a **`flag`**, as shown in the following code:

1. `deno run --allow-net args.ts deno land # Args [“deno”, “land”]. allow-net is a flag.`

The **`--allow-net`** flag passed after **`args.ts`** script is considered as an **argument**, as shown in the following code:

1. `deno run args.ts name --allow-net # Args[“name”, “--allow-net”]. Allow-net is not a flag`

File watcher

Deno will run the script in watch mode if **`--watch`** option is provided. It will rerun the script whenever a change is detected in the script. Watch mode watches for changes in the script and all its imports. This feature is still in unstable mode, which means it is not suitable for production use and will likely have breaking changes. For such features, we will have to use the **`--unstable`** flag along with **`--watch`** flag:

1. `# Watch welcome.ts and rerun if changes detected`
2. `deno run --watch --unstable welcome.ts`

Permission flags

Unlike Node, Deno does not provide access by default to the network, input/output, unless asked explicitly via permission flags. This functionality makes Deno very secure compared to Node. The following table includes all the major permission flags:

Permission	Description	Command
`--allow-net`	Allows network access	`deno run --allow-net example.ts`
`--allow-read`	Allows read operation	`deno run --allow-read example.ts`
`--allow-write`	Allows write operation	`deno run --allow-write example.ts`
`--allow-run`	Allows running subprocesses	`deno run --allow-run example.ts`
`--allow-env`	Allows environment access	`deno run --allow-env example.ts`
`--allow-plugin`	Allows use of plugins	`deno run --allow-plugin example.ts`
`--allow-all`	Allows all permissions	`deno run -A example.ts # -A = --allow-all`

Table 4.1: *Permission flags*

Permissions allow-list

By default, flags such as allow-write and allow-net provide access to any path or domain. To limit access to a specific domain/path, we use a permissions allow-list. Only mentioned paths in the list are allowed access to the resource.

Create a file named **`permission_list.ts`** and add code that uses the fetch API to fetch resources from sites *deno.land*, *github.com*, and *google.com*, respectively, as shown in the following code:

```
// permission_list.ts
fetch(“https://deno.land”);
fetch(“https://github.com”);
fetch(“https://google.com”); // Throws error if not added in permission_list
```

We would not get any network access error as all domains are allowed when we run the **`permission_list.ts`** script with **`--allow-net`** flag without permission list, as shown as follows:

```
# All domains are allowed to access the network
$ deno run --allow-net permission_list.ts
```

If we run the **`permission_list.ts`** script with the permission list allowing network access *deno.land* and *github.com* only, network access to all other domains will not be allowed, as shown in the following code:

```
# Only ‘deno.land’ and ‘github.com’ domains are allowed network access
$ deno run --allow-net=deno.land,github.com allow-net_example.ts
```

We used the domain for the **`allow-net`** permission list. Similarly, we can provide allowed paths for **`allow-read`** and **`allow-write`** flags. The following code reads previously created file **`args.ts`** and requires an **`allow-read`** flag. It uses **`readTextFile`** function from Deno API to read the **`args.ts`** file contents as utf8 encoded string.

```
// allow-read-list_example.ts
const argsContent = await Deno.readTextFile(“./args.ts”);
console.log(argsContent);
```

If we run the preceding code with the following command, it would log the contents of **`args.ts`** to console:

```
# Only ‘args.ts’ file is allowed to be read
$ deno run --allow-read=args.ts allow-read-list_example.ts
```

The following command would throw **`PermissionDenied`** error as **`args.ts`** is not added in the permission list:

```
1. # Only ‘args1.ts’, ‘args2.ts’ files are allowed to be read
2. $ deno run --allow-read=args1.ts,args2.ts allow-read-list_example.ts
```

Deno runtime APIs

Deno provides several number of runtime API functions that are useful during application development. All these runtime functions are documented at doc.deno.land (**https://doc.deno.land/https/github.com/denoland/deno/releases/latest/download/lib.deno.d.ts**). These functions can be categorized into two sections, namely, Web (browser) APIs and Deno global APIs.

Web APIs

Deno focuses on using existing Web standard APIs instead of creating new ones. Browser APIs such as **`fetch`**, **`setTimeout`**, and many other Web APIs are available in Deno. This gives an advantage that the developer familiar with web APIs can use them directly. We will list these APIs in the following table:

Web API	Description
`Fetch`	Used to make HTTP requests and fetch a resource from the network. It returns a promise.
`addEventListener`	Adds an event listener in the global scope.
`removeEventListener`	Removes **`eventListener`** added by the **`addEventListener`** function from the global scope
`Alert`	A standard alert function that shows an alert message and waits until Enter key is pressed
`setInterval`	Calls provided function repeatedly after provided interval until it is cleared using **`clearInterval`** function
`clearInterval`	Cancels an action initiated by **`setInterval`**
`setTimeout`	Invokes function after the specified time period

`clearTimeout`	Cancels delayed action initiated by `setTimeout`
`Btoa`	Creates base-64 ASCII encoded string from the provided string
`Atob`	Decodes string of data encoded by base-64 encoder like **btoa**
`Confirm`	Shows provided message and wait for user input. Returns user input in **boolean** (only **y** and **Y** are true)
`Prompt`	Shows provided message and wait for user input. Returns user input in **string** format
`dispatchEvent`	Dispatches event in the global scope and invokes registered listeners. Returns **false** for a cancelable event or if a listener called `Event.preventDefault()`

Table 4.2: *Web APIs*

Deno global APIs

All non-Web standard APIs are made available via Deno global namespace. APIs for i/o (`Deno.readFile`, `Deno.writeFile`) and networking (`Deno.listen`, `Deno.connect`) are available in this namespace.

Note that many of the functions have synchronous versions available.

For example, `create` function has a synchronous method `createSync.`

We will list the Web APIs and their synchronous versions (if available) in the following table:

Deno API	Sync	Description
`listen`	–	Used to create a listener with options for hostname, port, and transport
`close`	–	Closes previously opened resource ID (`rid`)
`connect`	–	Connects to the provided hostname, port
`copy`	–	Copies from source to destination

cwd	**–**	Returns current working directory (**cwd**) as a string
run	**–**	Spawns a new subprocess. The subprocess command must be defined as an array to the cmd key
exit	**–**	Exits the Deno process. An optional error code can be provided as an argument(default=0)
open	**openSync**	Open a file to perform read, write, and create operations
copyFile	**copyFileSync**	Copies contents and permissions of one file to another file
create	**createSync**	Creates a file if none exists or truncates the existing one
mkdir	**mkdirSync**	Creates a new directory with a provided string path
readDir	**readDirSync**	Reads directory at the provided path and returns **Deno.dirEntry** iterable. Error is thrown if the path is not a directory
readFile	**readFileSync**	Reads file as an array of bytes. The bytes can be converted to a string using TextDecoder. Returns an empty array if the path contains a directory
readTextFile	**readTextFileSync**	Reads file as an utf8 encoded string. Throws an error if the path contains a directory
remove	**removeSync**	Removes file or directory at the provided path. Throws error for a non-empty directory(with recursive option is not set to true), the path not found, permission denied
rename	**renameSync**	Renames/Moves resource (file or directory) from old path to new path. Overwrites a new path if it already exists
chmod	**chmodSync**	Changes permissions of provided file or directory using the three octal sequence called mode

`chown`	`chownSync`	Changes owner of file or directory (not available on Windows OS)
`writeFile`	`writeFileSync`	Writes data (Array of bytes encoded by TextEncoder) at the provided path by either creating a new file or overwriting the existing one
`writeTextFile`	`writeTextFileSync`	Writes data (string) at the provided path by either creating a new file or overwriting the existing one

Table 4.3: *Deno APIs*

Variables

Deno provides variables for various applications. It includes variables for CLI arguments, build-related info, environment variables, and so on. The following table lists major variables in the Deno runtime.

Variable	Description
`args`	Returns array of arguments in string. Arguments are provided via CLI
`build`	Provides system-related info such as `os`, `target`, `arch`, `vendor`, and `env`
`env`	Provides functions to `set`, `get`, and `delete` environment variables. `toObject` function lists all the available environment variables
`mainModule`	Returns URL of the entrypoint module in a string
`pid`	Returns the current process id of the runtime
`stdin`	A handle for `stdin` (Standard Input)
`stdout`	A handle for `stdout` (Standard Output)
`stderr`	A handle for `stderr` (Standard Error)
`version`	Provides version-related info for Deno, Typescript, and v8

Table 4.4: *Deno variables*

Classes

There are several classes in Deno runtime for different purposes. We will be using **`Request`** and **`Response`** classes in our upcoming projects. The following list contains the major classes with the description:

Class	Description
`Blob`	A file-like object containing raw, immutable data.
`Event`	Class representing **DOM Event**. Contains target attribute, **`PreventDefault()`** method, and so on.
`File`	Provides file-related info like **`lastModified`**, **`name`**, **`type`**, **`size`**, and so on. It is inherited from the **`Blob`** class.
`FormData`	Represents form fields in key-value pair format. Provides methods such as **`get`**, **`set`**, **`keys`**, **`values`**, **`entries`**, and so on.
`Headers`	Represents header-related data in object format.
`Location`	Provides location/URL of the linked object via **`globalThis.location`**
`Request`	Represents a request related info
`Response`	Represents a response to the request
`TextEncoder`	Provides utf8 encoding functionality useful to encode a string to an array of bytes
`TextDecoder`	Provides decoding function to decode an array of bytes to string
`URL`	Provides object with static methods required to create object URLs
`WebSocket`	Provides APIs to create and manage WebSocket connections and send and receive data

Table 4.5: *Deno classes*

Deno modules

The runtime ecosystem cannot be completed without the modules (standard and third-party). In fact, the popularity of

Node is owed to a large number of third-party modules available via npm. There are two types of modules available in Deno: Standard Library and Third-Party modules. We will check both types of modules in the following sections.

Standard library

Deno standard library is a collection of standard modules and is inspired by the standard library of the Go language. All the standard modules are reviewed by Deno Core Team and do not have external dependencies. These modules are available at **https://deno.land/std**. We have listed them in the following table:

Module	Description
`archive`	Module providing **Tar**, **Untar** functionality.
`Async`	Asynchronous task utility with **deferred** (Promise with reject, resolve functions)**,** **delay** (delay promise resolve) functionality.
`Bytes`	Provides helper functions to manipulate bytes.
`Datetime`	Datetime module with **parse** (string to date), **format** (date to string), **difference** (the difference between 2 dates), and some additional functions.
`Encoding`	Ported from Go’s **encoding** module, this module is useful for dealing with external data structures such as csv, yaml, toml, and so on
`flags`	A command-line argument parser based on `minimist` npm module. It is useful for CLI tool development.
`Fmt`	Provides i/o formatting functions like `printf`**.**
`Fs`	Provides helper functions for file system manipulation. Provides `emptyDir`, `ensureDir`**,** `exists`**,** `move`**,** `copy`**,** `walk`**,** and other functions. This module is unstable as of now, and hence, will need to use `–unstable` flag during usage.
`Hash`	Used to create a hash with a large number of supported algorithms.

`http`	Useful for creating HTTP Web servers. It also includes a file server function to serve local files over HTTP.
`Io`	Provides functions like **`Reader`, `Writer`, `Streams`,** and so on.
`Log`	A logger module with log levels.
`Mime`	Used to deal with multipart form data.
`Node`	The Compatibility module for Node's Standard Library. Enables use of Node modules like lodash in Deno.
`Path`	Provides path related functions.
`Permissions`	Provides Permissions to the scripts.
`Signal`	Provides APIs to capture and monitor OS signals like SIGINT.
`Testing`	Provides test-related utilities such as **assert, equal,** and so on.
`Textproto`	Inspired by Go's **`textproto`** module.
`Uuid`	Provides functions to **generate** and **validate** **Universal unique identifier** (**UUID**) with support for versions 1, 4, and 5.
`Wasi`	Provides the implementation for **WebAssembly System Interface** (**WASI**)
`ws`	Useful to create WebSocket servers.

Table 4.6: *Deno standard library*

Third-party modules

Deno 1.0 was released in May 2020. Since then, there have been a lot of modules already available for different purposes, such as Web servers (**`oak`**), authentication (**`onyx`**), databases (**`mongo`**), ORMs (**`denodb`**), JWT (**`djwt`**), and so on. We are going to have a brief overview of many of such modules in the following table:

Third-Party Module	Description
oak	An extremely popular web middleware framework,

	including middlewares like routing. It is inspired by **koa** framework.
Aleph	A react framework inspired by **Next.js** that supports **Server-Side Rendering (SSR)**, **Static Site Generator** (**SSG**)**,** and so on.
Pagic	A React framework for Static Site Generation
drash	A lightweight REST MicroFramework that supports API Development and **Single Page Applications** (**SPA**) development using React and Vue.
Servest	A web server compatible with **std/http**, it supports WebSocket and JSX out of the box.
Denon	A Deno alternative to Node's **nodemon** that restarts the app or server automatically whenever changes are detected. It also serves as a replacement wrapper for `deno` command.
Webview	A cross-platform module for creating desktop GUI apps.
Denodb	ORM supports MySQL, PostgreSQL, MongoDB, MariaDB, and SQLite databases.
Trex	An NPM-like module manager that aligns with Deno's philosophy. It uses `import_map.json` as a module import controller.
Velociraptor	An alternative to npm scripts, it supports scripts written in **yaml**, **json,** and **ts** format.
Postgres	A driver module for the PostgreSQL database.
Redis	A Redis client module for Deno.
Dvm	A Deno version manager is similar to `nvm`.
Mongo	A Mongo driver module for MongoDB database.
Vno	A compiler and bundler module for Vue components.
Obsidian	A GraphQL client and server module with caching support.
Eta	A lightweight JavaScript Template Engine is written in TypeScript.
Mysql	Driver module for MySQL database.
Sqlite	Driver module for SQLite database.

Ssgo	**Static Site Generator** (**SSG**) module built with Deno.
Onyx	Authentication module inspired by passport.js, which is compatible with **oak**.
Otpauth	OTP authentication module.
Aqua	A web framework module that is fast and minimal.
Denox	A cross-platform script runner module.
Dejs	EJS Template Engine module.
Dotenv	A module to handle environment variables.
Udd	A module to update Deno dependencies to the latest versions.
Djwt	A module to use **JSON Web Token** (**JWT**) in Deno.
Cotton	A SQL Database Toolkit module.
Denomander	A **command-line interface** (**CLI**) tool module.
Lume	A Static Site Generator module for Deno inspired by Jekyll.
Oak_graphql	A GraphQL middleware module for **oak** framework.
Drake	A Task Runner module inspired by **make, rake,** and **jake.**
rhum	A Testing framework module for Deno which is lightweight from the creators of **drash.**
websocket	A WebSocket module for Deno.
dext.js	A Preact framework module inspired by `next.js`.
denoliver	A static file server module with live reload support.
wocket	A WebSocket module from the creators of **drash**.
dem	A module version manager for Deno.
bundler	A bundler module to transpile TypeScript code for the web.
dmm	A Deno module manager from the creators of **drash**.
opencv	A pre-compiled OpenCV to JavaScript + WebAssembly module.

garn_validator	A validator module with many validation functions.
watch	A file watcher module for Deno.
smtp	A **Simple Mail Transfer Protocol** (**SMTP**) module for sending and receiving mails.
nanoid	A NanoID implementation module for Deno.
superoak	HTTP Assertions module for **oak** framework.
dpx	NPX like module runner (runs module without installing) for Deno.
drash_middleware	A middleware module for **drash** framework from the creators of **drash**.
cors	A **Cross-Origin Resource Sharing** (**CORS**) module for Deno.
postcss	A Postcss module for Deno.
oak_middleware	A middleware collection module for **oak** framework from the creators of **oak**.

Table 4.7: *Deno third-party modules*

Module registries

Module registries are places where the Deno modules are hosted. Just like **`npm`** in Node, Deno has several module registries built using modern features like blockchain.

After listing the standard and third-party modules for Deno, we will be checking the major module registries available for Deno:

Module Registry	Description
`deno.land/x`	An official module registry hosting large numbers of third-party modules (1651 modules at the time of writing)
`nest.land`	A modern Deno module registry built using **blockweave,** a derivative of **blockchain**. The published modules are immutable, decentralized, and permanent, meaning they can never be deleted.
`Skypack CDN`	A CDN to host pre-optimized ES modules offered by

	Pika. It provides minification, caching, polyfilling, and converts npm packages to ES modules. Whenever **?dts** is appended to the module URL, it automatically fetches type declarations using the **X-TypeScript-Types** header.
`esm.sh`	A CDN for ES modules from the creator of the **aleph** framework that bundles all dependencies (except peer dependencies) for every module. Node packages are transformed to ES modules using **esbuild** to run on Deno via polyfill. It provides type declarations via the **X-TypeScript-Types** header by default.
`jspm`	A module CDN that converts npm packages to ES modules.
`Web`	Any ECMAScript module hosted on the internet (e.g., GitHub) can be used via ESM import supported by Deno.

Table 4.8: *Deno module registries*

Central dependencies in deps.ts

There is one convention in Deno land to import all the dependencies in the **`deps.ts`** file. These dependencies are re-exported and used throughout the project. This helps us maintain all the dependencies in a single place. Similarly, development dependencies are kept in a single file named **`dev_deps.ts`**.

In the following code, we are importing all the dependency URLs in the **`deps.ts`** file and exporting them to use across the project. We are importing the **`titleCase`** function from a third-party module case that returns each word in the title case. We are also importing **`assertEquals`** function from the testing module from Deno’s standard library.

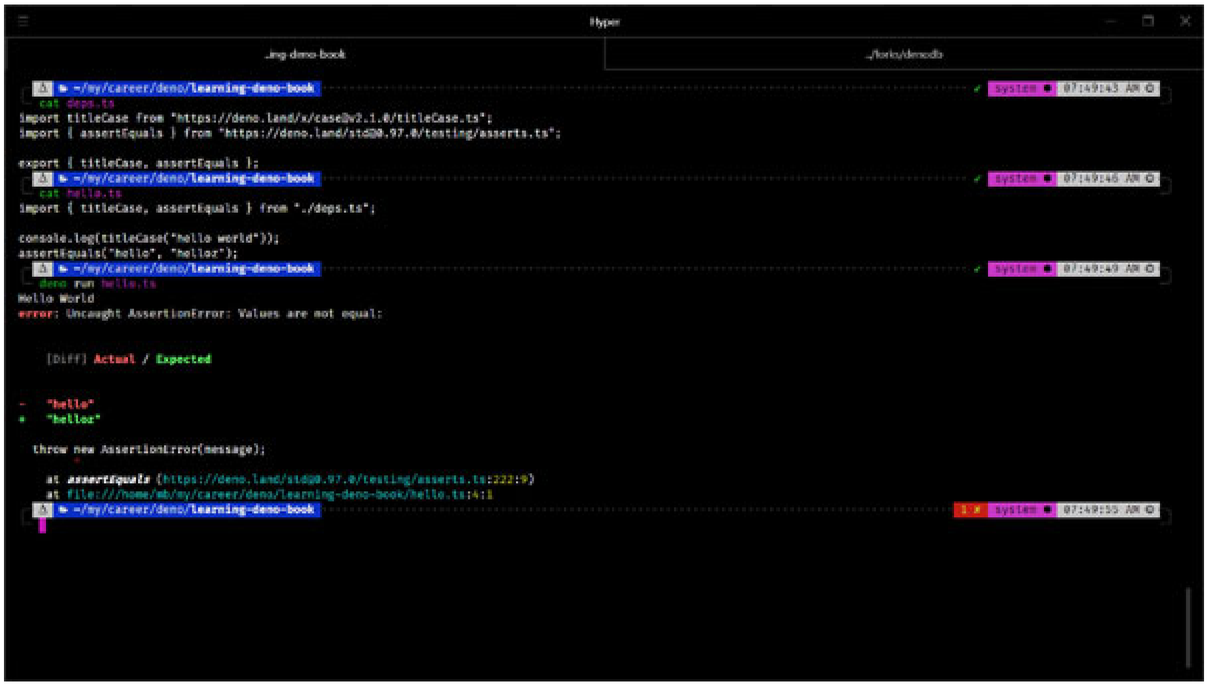
```
// deps.ts
import titleCase from “https://deno.land/x/case@v2.1.0/titleCase.ts”;
import { assertEquals } from “https://deno.land/std@0.97.0/testing/asserts.ts”;
```

```

export { titleCase, assertEquals };
```

We will now import the modules from **`deps.ts`** file in **`hello.ts`** file, as shown in the following code:

```
// hello.ts
import { titleCase, assertEquals } from "./deps.ts";

console.log(titleCase("hello world"));
assertEquals("hello", "helloz");
```

The preceding code logs **`"hello world"`** string in title case **`"Hello World"`** and the **`assertEquals`** function throws **`AssertionError`** as actual (**`"hello"`**) and expected (**`"helloz"`**) values are not equal, as shown in the following figure:



```
~/my/career/deno/learning-deno-book                              ✓ system ● 07:49:43 AM
cat deps.ts
import titleCase from "https://deno.land/x/case@v2.1.0/titleCase.ts";
import { assertEquals } from "https://deno.land/std@0.97.0/testing/asserts.ts";

export { titleCase, assertEquals };
~/my/career/deno/learning-deno-book                              ✓ system ● 07:49:46 AM
cat hello.ts
import { titleCase, assertEquals } from "./deps.ts";

console.log(titleCase("hello world"));
assertEquals("hello", "helloz");
~/my/career/deno/learning-deno-book                              ✓ system ● 07:49:49 AM
deno run hello.ts
Hello World
error: Uncaught AssertionError: Values are not equal:


    [Diff] Actual / Expected


-   "hello"
+   "helloz"

  throw new AssertionError(message);
        ^
    at assertEquals (https://deno.land/std@0.97.0/testing/asserts.ts:222:9)
    at file:///home/mb/my/career/deno/learning-deno-book/hello.ts:4:1
~/my/career/deno/learning-deno-book                          1 ✗ system ● 07:49:55 AM
```

Figure 4.2: *deps.ts example*

Conclusion

We are now concluding this chapter on the Deno Ecosystem. It outlined various sections of the Deno Ecosystem, such as runtime API, CLI, standard and third-party modules, module registries, and so on. Being a new technology in the industry, the Deno Ecosystem is evolving at a rapid pace. We would likely see the Deno community come up with new and exciting ideas and implementations in the coming time, as they have done until now.

With the end of this chapter, we are now ready to move to the most exciting part of the book—writing code. We will start with writing a Deno application from scratch in the upcoming chapters. Till then, keep exploring the Deno ecosystem.

CHAPTER 5
Todos App—API

Introduction

We are now ready to enter the Deno coding battlefield or playground (it depends on how you see it). We will create a **CREATE, RETRIEVE, UPDATE**, and **DELETE** (**CRUD**) application in this chapter. A Todos app. It will be a great way to begin our Deno coding journey. We will learn, tweak, and play around with the Todos app created using a Full Stack React framework called **`Aleph.js`** (aleph), listed in *Table 4.7* of *Chapter 4, Deno Ecosystem*. The API will be implemented in this chapter, whereas the UI will be implemented in the next one.

Structure

In this chapter, we will cover the following topics:

- Introduction to Aleph.js
    - Features
    - Concepts
        - Pages
        - Routing
        - APIs
- Todos app API implementation
    - Routes
        - GET (list)
        - POST

- PATCH
- GET (object)
- DELETE

Objective

We will be able to create a server-side API using the *Aleph.js* framework after studying this chapter. The aim of this chapter is to get familiar with Aleph.js and use its concepts to create API. Aleph.js is a Full Stack React framework used to create API and UI as well. We will start the chapter by introducing Aleph.js features and concepts. The API for the Todos app will then be developed using Aleph.js.

We will also learn to implement different **Representational State Transfer** (**REST**) API actions like **`POST`** (CREATE), **`GET`** (RETRIEVE), **`PATCH`** (UPDATE), and **`DELETE`** (DELETE) after studying this chapter.

Introducing Aleph.js

Aleph.js is a Full Stack React framework inspired by Next.js. It uses the ES module import syntax without the need for Webpack or other bundlers. It does not re-bundle every change but only the changed module and updates instantly in the browser by **Hot Module Replacement** (**HMR**) with React Fast Refresh.

Aleph.js can be installed using the following command:

```
deno install -A -f -n aleph https://deno.land/x/aleph/cli.ts
```

Features

Aleph.js is a feature-rich framework that provides server-side rendering, static site generation, API development, and many other features. Some of the major features are as follows:

- File-system-based pages and API routing
- API development
- Server-side rendering
- Static site generation
- Zero config
- Use of import maps
- HMR with Fast Refresh
- Access to Deno runtime within component via **`useDeno`** hook

Command Line Interface

Aleph.js **Command Line Interface** (**CLI**) provides commands for static site generation, creating a new app with boilerplate code, upgrading Aleph.js, and so on. All major commands are listed in the following table:

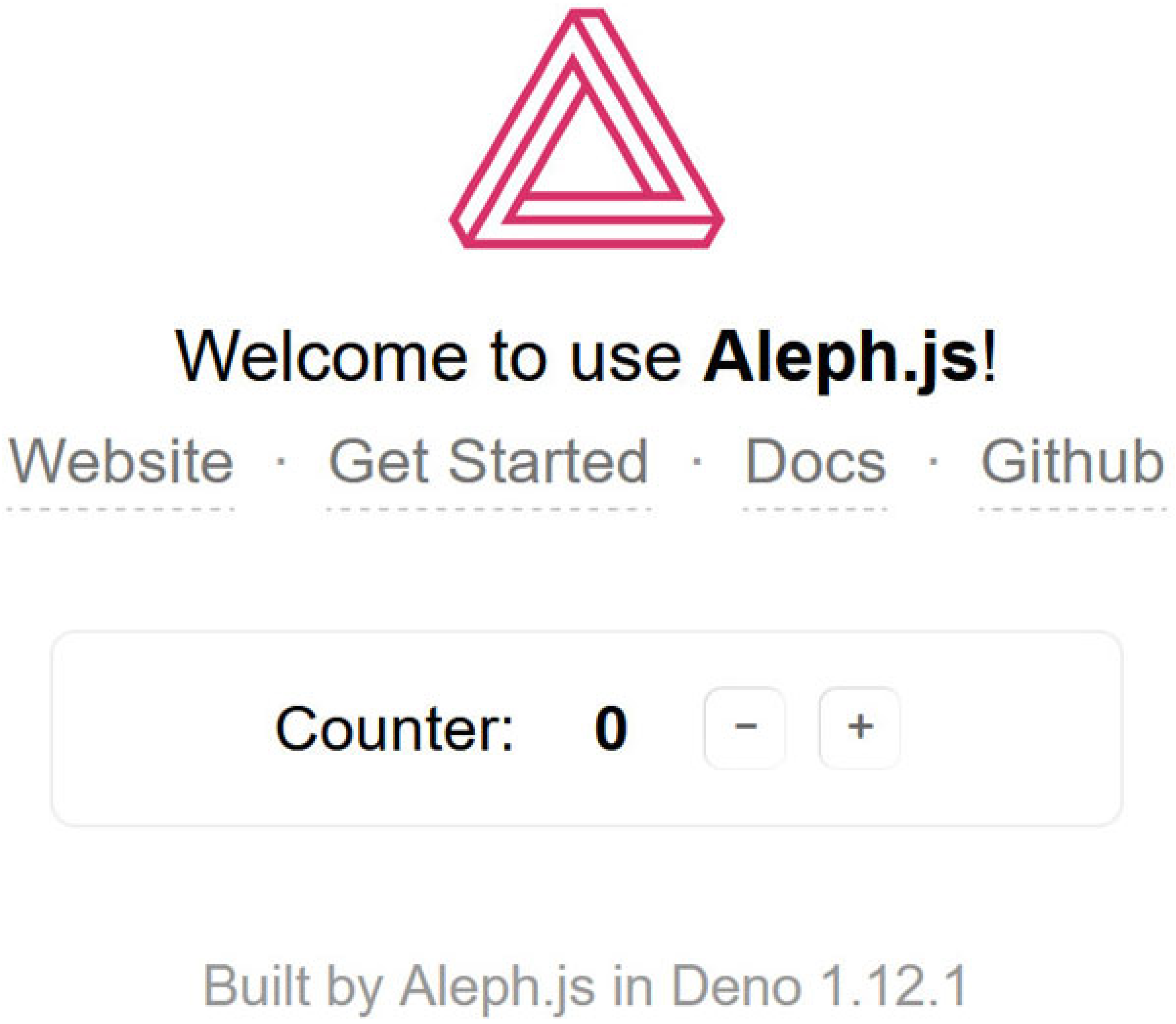
Command	**Description**
`aleph init`	Create new app with boilerplate code
`aleph upgrade`	Upgrade aleph.js
`aleph`	Prints all cli commands
`aleph -h / aleph --help`	Prints help message
`aleph -v / aleph --version`	Prints aleph.js version
`aleph dev`	Starts server in development mode
`aleph start`	Starts server in production mode
`aleph build`	Builds the app to static site
`aleph analyze`	Analyzes app dependencies

Table 5.1: *Aleph.js CLI*

`aleph init` command provides an example app with boilerplate code. We will be using it to initialize the demo app as follows:

1. `aleph init hello-aleph // Creates hello-aleph folder with boilerplate code`
2. `cd hello-aleph`
3. `aleph dev // Runs app in development mode`

You can visit **http://localhost:8080** to view the demo app, as shown in the following figure:


Welcome to use **Aleph.js**!

Built by Aleph.js in Deno 1.12.1

Figure 5.1: *Aleph.js Demo App*

Concepts

There are three basic concepts we need to learn to develop a full stack app using aleph. We will use these concepts for creating pages, routing based on filenames, and API.

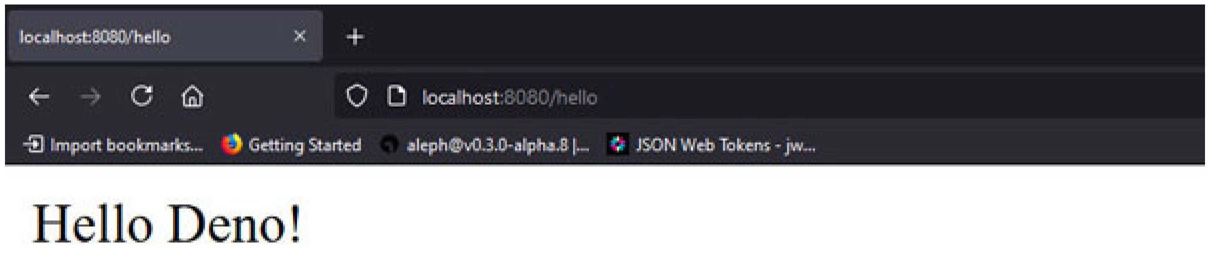
Pages

Any file with extension **`*.js`**, **`*.jsx`**, **`*.ts`**, **`*tsx`**, and **`*.mjs`** in /pages folder having a default exported React component is considered as a page. As aleph.js supports filename-based routing, each page will then be associated with a route based on the filename.

Example:

If we create **`pages/hello.tsx`** with the default exported React component as shown in the following code, then we can access that page at **http://localhost:8080/hello**, as shown in *figure 5.2*.

```
// pages/hello.tsx
import React from “react”;

export default function Hello() {
return “Hello Deno!”;
}
```



Hello Deno!

Figure 5.2: *Hello Deno! Page*

Every page is **pre-rendered**, meaning HTML for the page is generated in advance, resulting in better performance and SEO. Aleph.js also supports a **hydration** process, where associated JavaScript code for the HTML is run when the page is loaded by the browser, making it fully interactive.

Routing

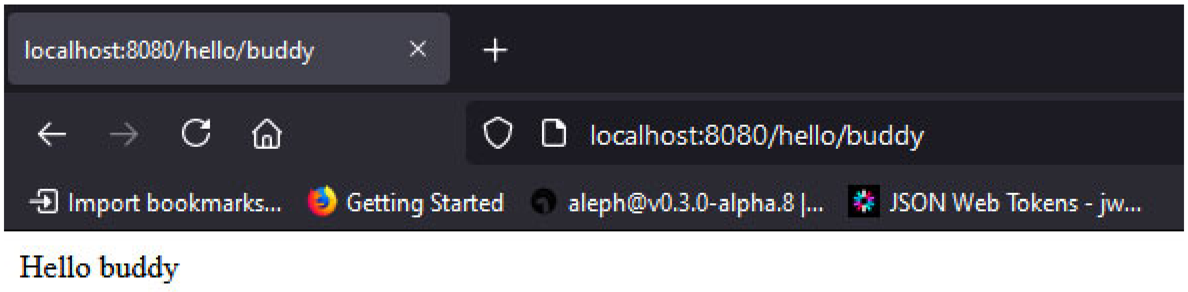
Aleph.js supports file-system-based routing. It automatically routes files named **`index`** to the root of the directory. **Dynamic Routes** are also supported via **bracket** syntax ([]) or **$** sign prefix.

```
// Index routing
pages/index.tsx => localhost:8080/
pages/hello/index.tsx => localhost:8080/hello

// Dynamic routing
// $ prefix
pages/hello/$name.tsx => pages/hello/buddy
```

The following code uses a **`params`** object of **`useRouter()`** hook to obtain the passed parameter value. The property key of **`params`** is determined by the filename ($name).

```
// pages/hello/$name.tsx
import React from “react”;
import { useRouter } from “https://deno.land/x/aleph/framework/react/mod.ts”;


export default function Hello() {
const { params } = useRouter();
return `Hello ${params.name}!`;
}
```

If we enter **`route /hello/buddy`** in the browser, where **`buddy`** is the value of the **`name`** parameter, we shall see the response shown in *figure 5.3*.


localhost:8080/hello/buddy
localhost:8080/hello/buddy
Import bookmarks...
Getting Started
aleph@v0.3.0-alpha.8 |...
JSON Web Tokens - jw...

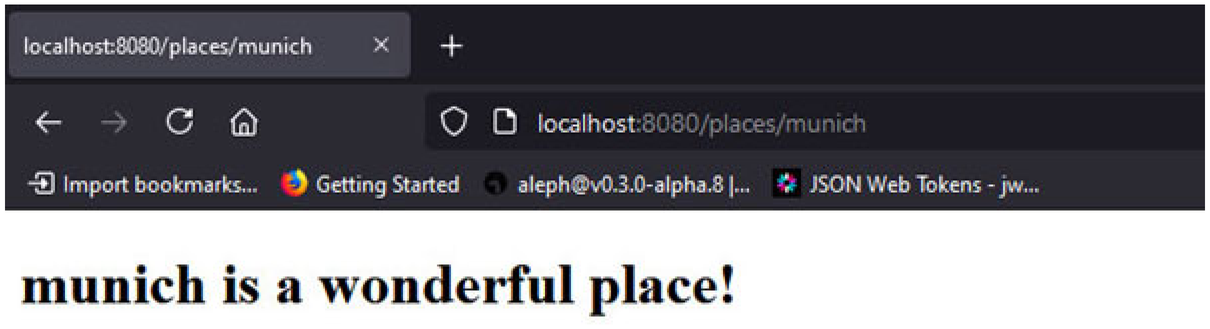
Hello buddy

Figure 5.3: *Dynamic routes ($ Syntax)*

```
// Bracket syntax []
pages/places/[place].ts => pages/places/munich
```

Similar to $ sign syntax, a bracket syntax [] does serve the same purpose as shown as follows. Here, the parameter key is the place (derived from the filename **`[place].ts`**). If we enter **`/places/munich`** in the browser, where Munich is the value of the place parameter, we shall see the response as shown in *figure 5.4*.

```
// pages/places/[place].tsx
import React from “react”;
import { useRouter } from “https://deno.land/x/aleph/framework/react/mod.ts”;

export default function Places() {
const { params } = useRouter();

return <h1>{`${params.place} is a wonderful place!`}</h1>;
}
```


localhost:8080/places/munich
localhost:8080/places/munich
Import bookmarks...
Getting Started
aleph@v0.3.0-alpha.8 |...
JSON Web Tokens - jw...

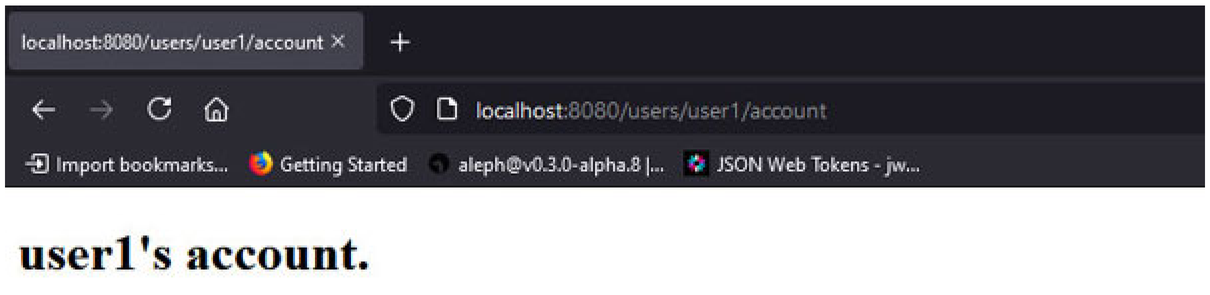
munich is a wonderful place!

Figure 5.4: *Dynamic routes (bracket syntax)*

Nested Routes are supported and can be achieved as shown in the following code:

```
pages/users/[username]/account.tsx =>
pages/users/user1/account
```

```
// pages/users/[username]/account.tsx
import React from “react”;
import { useRouter } from “https://deno.land/x/aleph/framework/react/mod.ts”;

export default function UserAccount() {
const { params } = useRouter();

return <h1>{`${params.username}’s account.`}</h1>;
}
```


localhost:8080/users/user1/account
localhost:8080/users/user1/account
Import bookmarks...
Getting Started
aleph@v0.3.0-alpha.8 |...
JSON Web Tokens - jw...

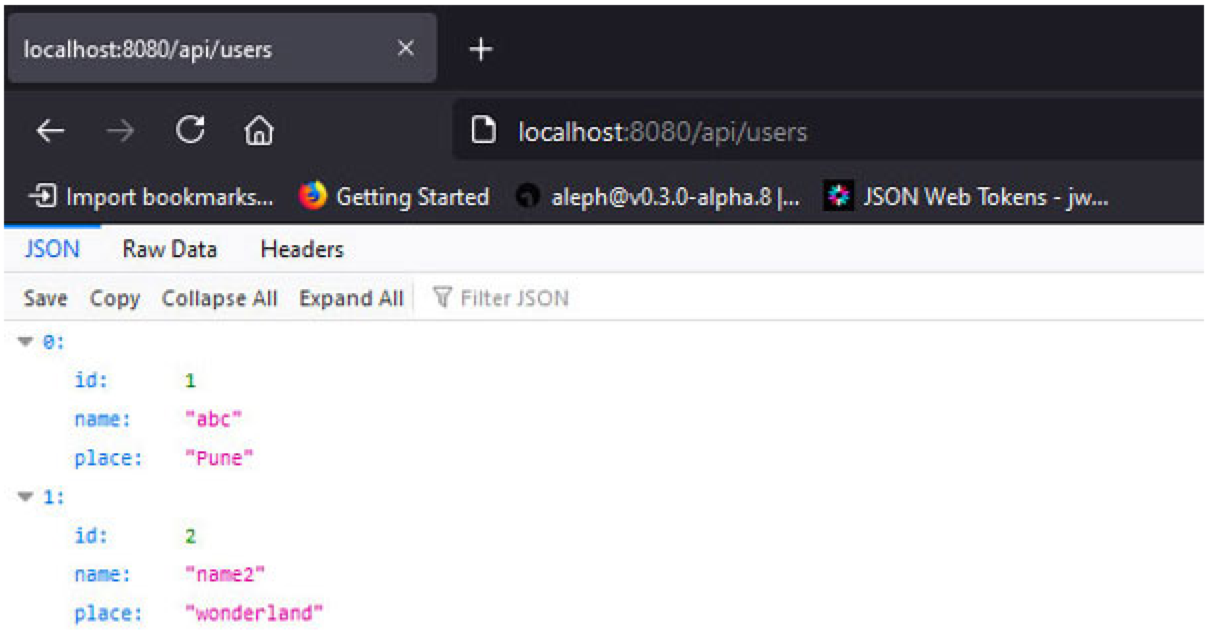
user1's account.

Figure 5.5: *Nested routes*

Application Programming Interface (API)

Similar to pages, any file with **`*.ts`**, **`*.js`**, **`*.mjs`** extension having default exported function inside **`api/`** folder is mapped to **`/api/*`** and is treated as an API endpoint. The exported function will be of type **`APIHandler`**.

```
// api/users/index.ts
import type { APIHandler} from “aleph/types.d.ts”;

const users = [
{ id: 1, name: “abc”, place: “Munich” },
{ id: 2, name: “name2”, place: “wonderland” },
];

export const handler: APIHandler = ({response}) => {
response.status = 200;
response.json(users);
}
```

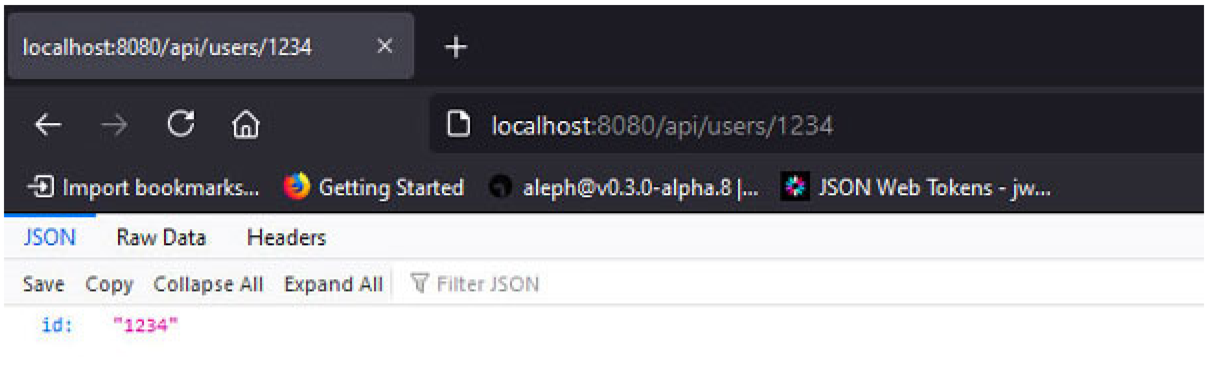


```
▼ 0:
    id:       1
    name:     "abc"
    place:    "Pune"
▼ 1:
    id:       2
    name:     "name2"
    place:    "wonderland"
```

Figure 5.6: *API routing*

Dynamic API routes are also supported. We can create a file like **`api/users/[id].ts`** with the default export function. The **`id`** can then be passed as a parameter like **`api/users/1234`** and will be obtained from the **`router.params`** object.

```
// api/users/[id].ts
import type { APIHandler } from “aleph/types.d.ts”;

export const handler: APIHandler = async ({response, router}) => {
response.json(router.params);
}
```



```
id: "1234"
```

Figure 5.7: *Dynamic API route*

Initializing Todos App

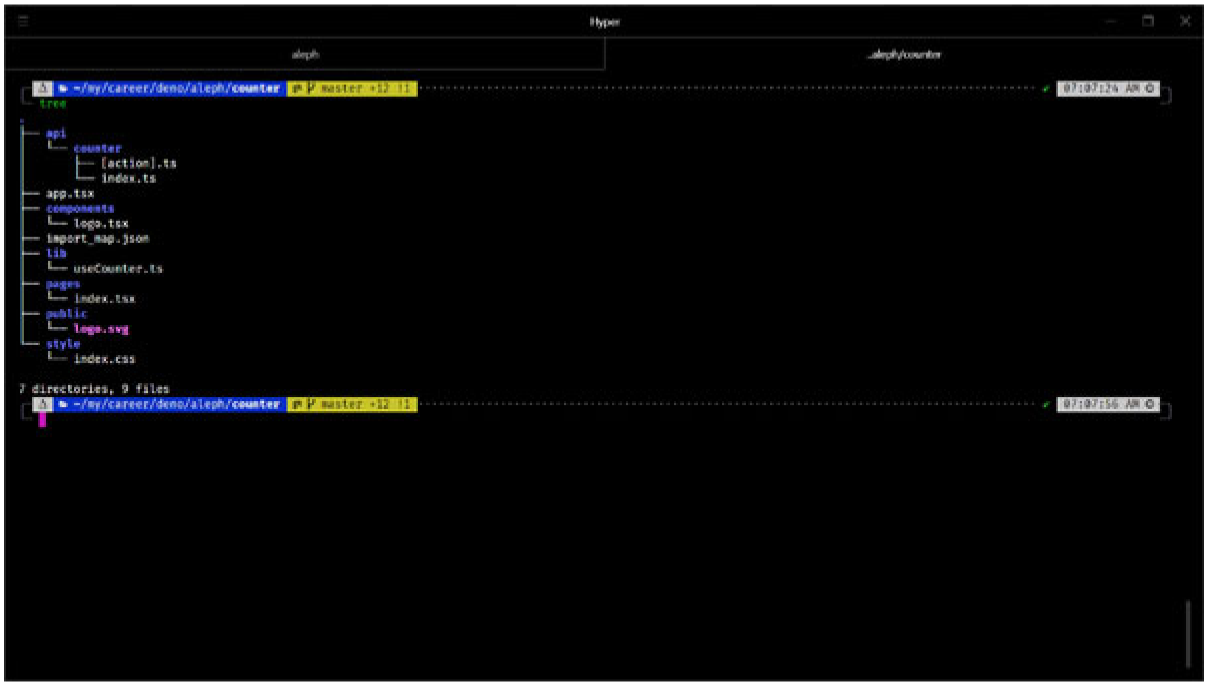
We have checked the major Aleph.js concepts required for full-stack application development. We will now initialize our todos app using the **`aleph init`** command. Head over to the terminal and type the following commands. Make sure you have installed Deno, Aleph, and VS code before running the commands.

```
aleph init todos-app
cd todos-app
```

3. `code . // Make sure VS code is installed`
4. `aleph dev // Server running on http://localhost:8080`

Project structure

The project structure shown in *figure 5.8* is familiar to us. We have already seen API and page concepts. The **`app.tsx`** file is the entry point for our app. **`styles`** folder contains styling-related files, the components folder contains React Component files, the public folder contains all the static files, and **`lib`** folder contains custom hooks and other utility functions.



```
~/my/career/deno/aleph/counter  master +12 !1                    07:07:24 AM
tree
.
├── api
│   └── counter
│       ├── [action].ts
│       └── index.ts
├── app.tsx
├── components
│   └── logo.tsx
├── import_map.json
├── lib
│   └── useCounter.ts
├── pages
│   └── index.tsx
├── public
│   └── logo.svg
└── style
    └── index.css

7 directories, 9 files
~/my/career/deno/aleph/counter  master +12 !1                    07:07:56 AM
```

Figure 5.8: *Aleph project structure*

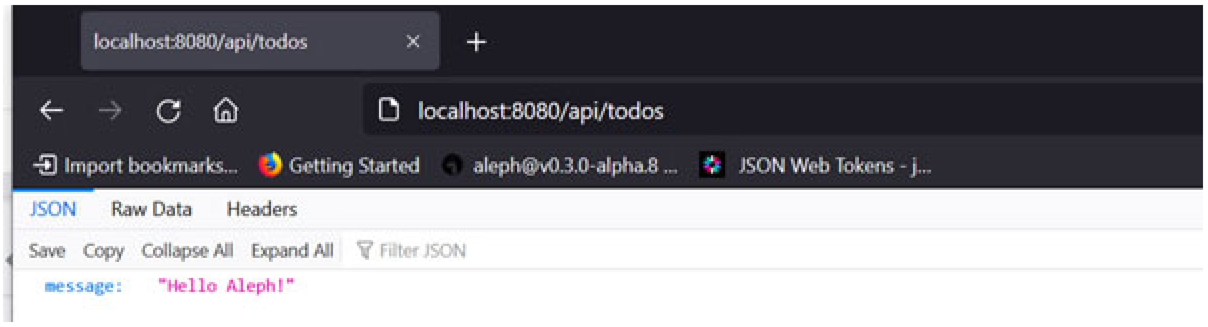
Our development approach will be creating API endpoints first and then creating UI that consumes the API.

Todos API

Before creating the UI for Todos App, we will need to create the API endpoints for CRUD actions. Create an **`index.ts`** file in a folder called **`todos`** in the **`api`** folder. It should have a path like **`api/todos/index.ts`**.

```
// api/todos/index.ts
import type { APIHandler} from “aleph/types.d.ts”;

export const handler: APIHandler = ({response}) => {
response.json({ message: “Hello Aleph!” });
}
```

Preceding code exports a function of type **`APIHandler`**. If you hit **http://localhost:8080/api/todos** using Postman, curl, or browser, the function will be called with the context argument containing request, response, and router objects. The **`response.json`** function is then used to reply with JSON content, as shown in *figure 5.9*.



```
message:    "Hello Aleph!"
```

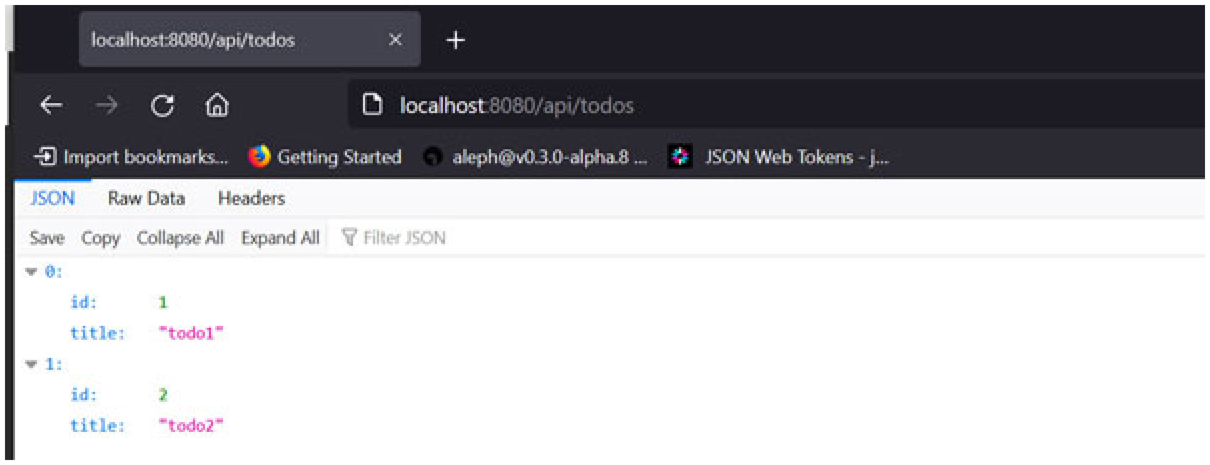
Figure 5.9: ***`api/todos`*** *endpoint*

GET (list)

Now, we should modify the file to GET all existing todos instead of the message. We will create a Todos array with dummy todos, as shown here:

```
// api/todos/index.ts
import type { APIHandler} from “aleph/types.d.ts”;

const todos = [
{ id: 1, title: “todo1” },
{ id: 2, title: “todo2” },
];

export const handler: APIHandler = ({request, response})
=> {
switch (request.method) {
case “GET”:
   response.json(todos);
   break;

  default:
   response.status = 405;
   response.json({ message: “Action not allowed” });
   break;
}
}
```

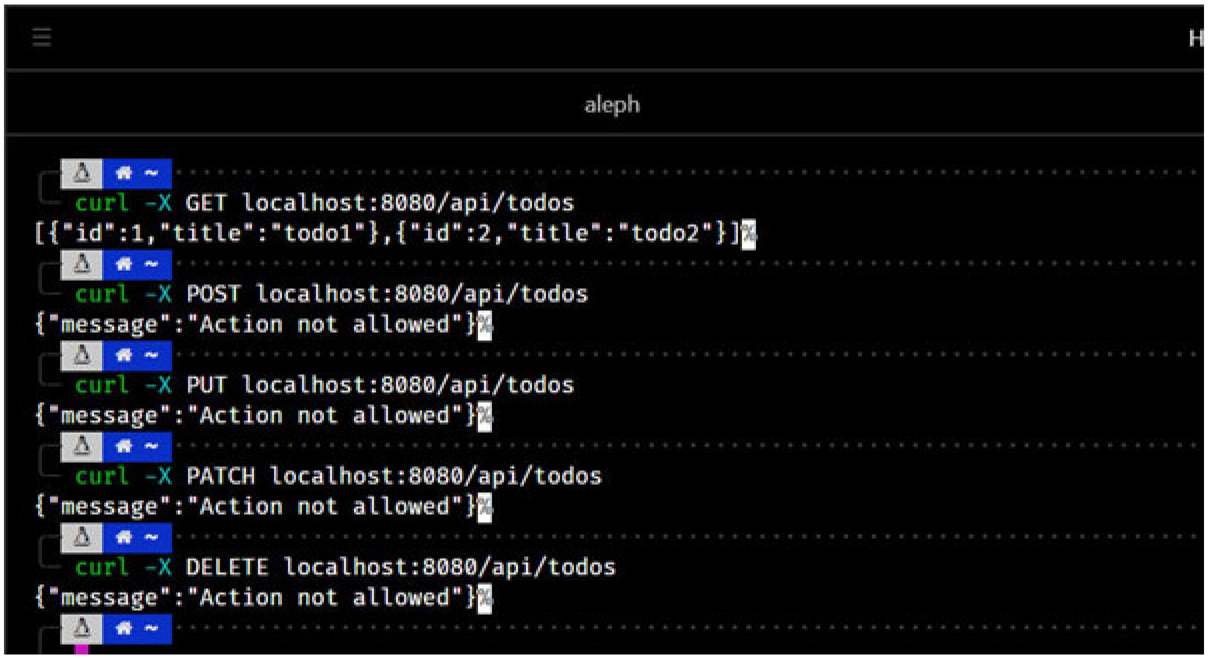
Here, we are using a **`switch`** statement with the request **`method (GET, POST)`** as a case.



```
▼ 0:
    id:      1
    title:   "todo1"
▼ 1:
    id:      2
    title:   "todo2"
```

Figure 5.10: *GET/api/todos endpoint*

It will respond with all the existing todos for the **`GET`** method (*figure 5.10*), and if the requested action is not defined in the switch statement, it will reply with the **`405`** Method Not Allowed code (*figure 5.11*). The break skips the following cases and ensures that only matched case code is executed.



```
curl -X GET localhost:8080/api/todos
[{"id":1,"title":"todo1"},{"id":2,"title":"todo2"}]%
curl -X POST localhost:8080/api/todos
{"message":"Action not allowed"}%
curl -X PUT localhost:8080/api/todos
{"message":"Action not allowed"}%
curl -X PATCH localhost:8080/api/todos
{"message":"Action not allowed"}%
curl -X DELETE localhost:8080/api/todos
{"message":"Action not allowed"}%
```

Figure 5.11: *405 code for undefined actions*

POST

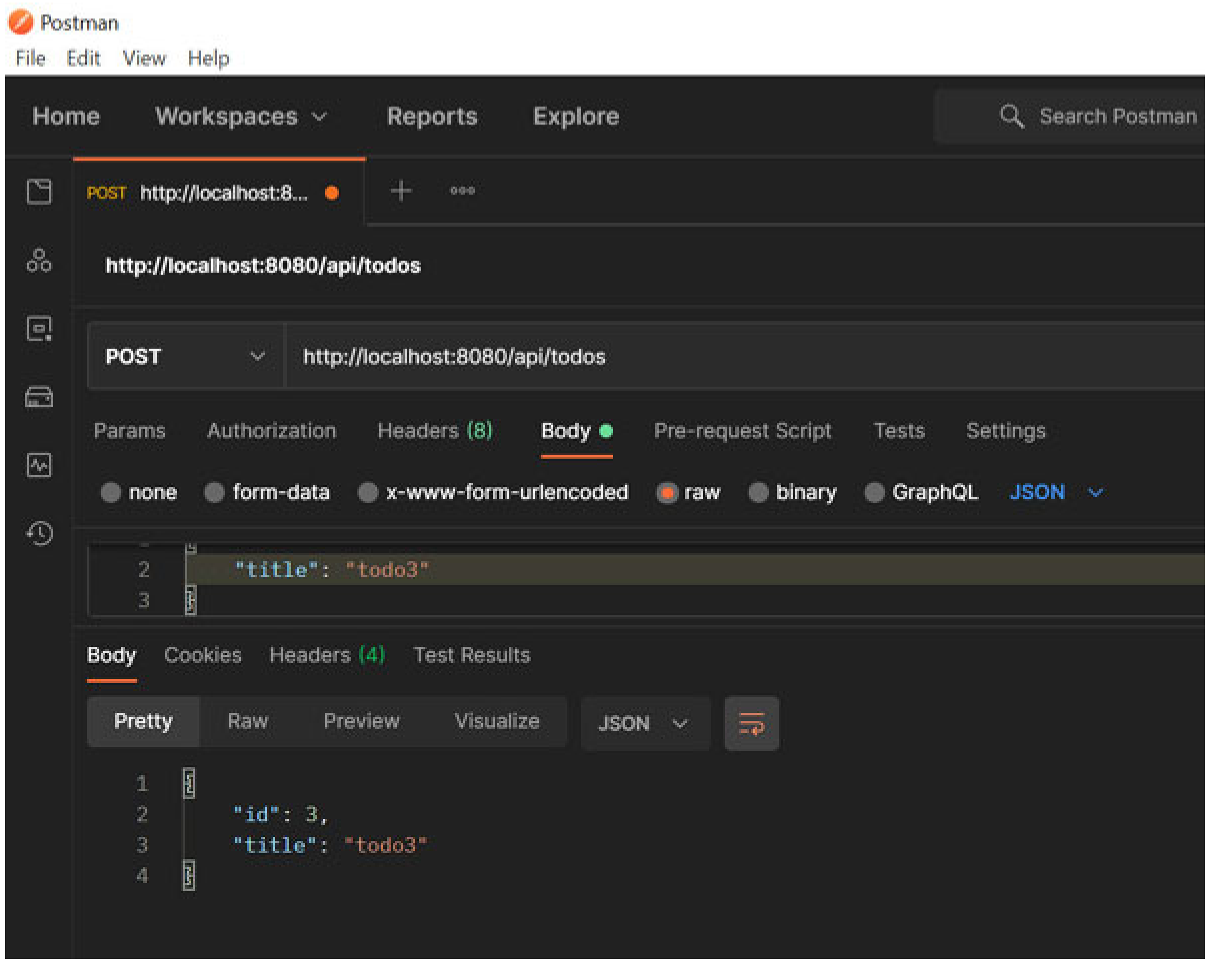
We will now implement **`CREATE (POST)`** action for adding todo.

```
…
export const handler: APIHandler = async ({request, response}) => {
switch (request.method) {
  case “GET”:
   response.json(todos);
   break;

  case “POST”:
```

```
  let body;

  try {
   body = await request.json();
  } catch (error) {
   response.status = 400;
   response.json({ error: error.message });
   break;
  }

   const title = body.title ? body.title.toLowerCase() : null;
  if (!title) {
   response.status = 400;
   response.json({
    message: “title not provided”,
   });
   break;
  }

    let todo = todos.find((_todo) => _todo.title === title);

  if (!todo) {
   todo = {
    id: todos[todos.length - 1].id + 1,
    title, // Shorthand for title: title
   };

```

```
      todos.push(todo);
    }
    response.json(todo);
    break;

  default:
…
```

We are decoding the request body using the **`request.json()`** method that parses the request body as JSON. **`400`** code is responded with an error message for invalid or no JSON and also if the title is not provided. We are using the **`toLowerCase()`** function to get todo titles in lowercase. This will help us find existing Todo with the same title using the find method. If an existing todo is not found, a new todo will be created and added to the Todos list. The **`id`** property set uses the id of the last todo item (last Todo id + 1). The Todo (existing or created) will then be returned using the **`response.json`** function. Now, try adding a todo, as shown in *figure 5.12*.

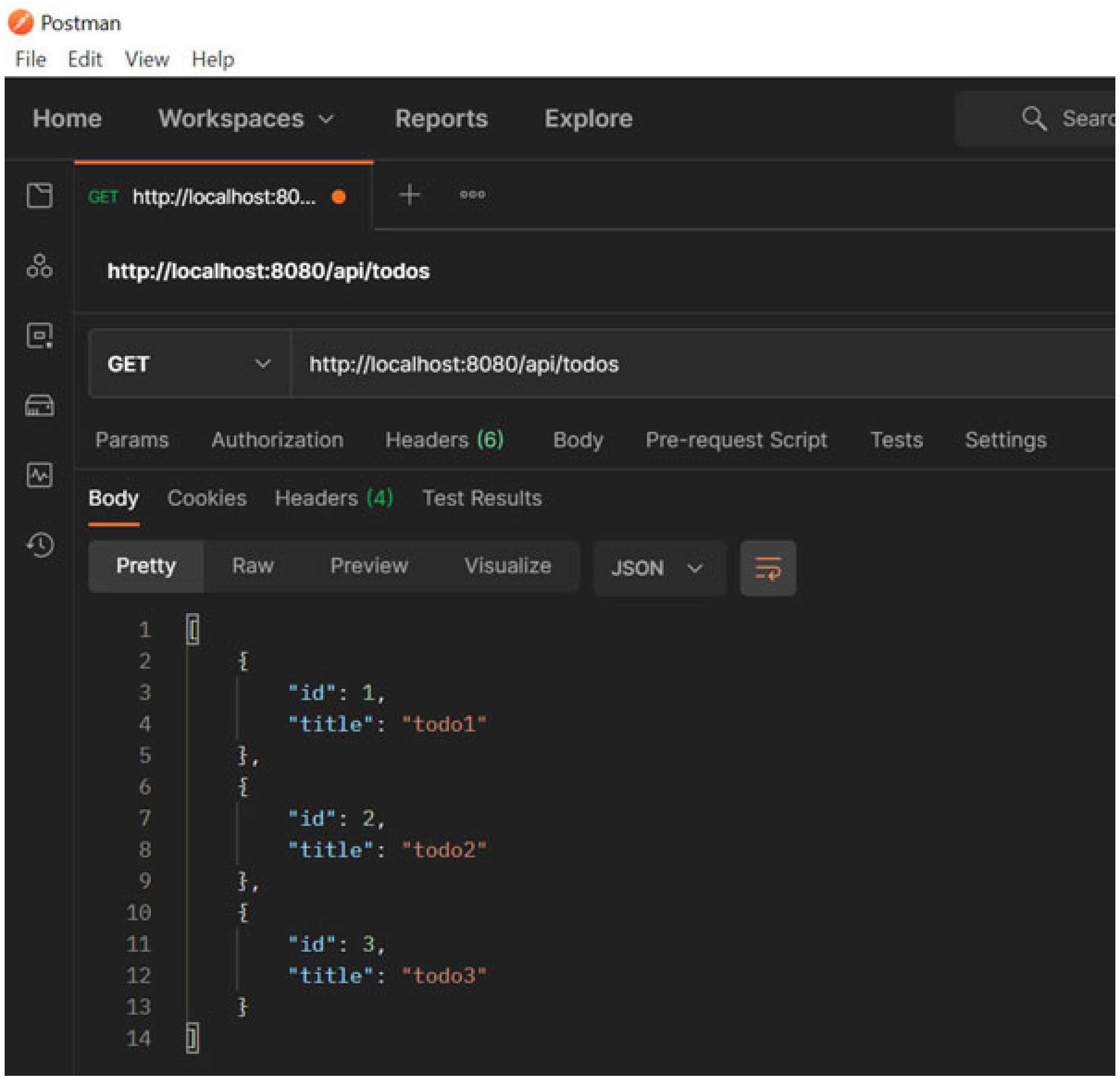

Postman
File Edit View Help
Home Workspaces Reports Explore
Search Postman
POST http://localhost:8...
http://localhost:8080/api/todos
POST
http://localhost:8080/api/todos
Params Authorization Headers (8) Body Pre-request Script Tests Settings
none form-data x-www-form-urlencoded raw binary GraphQL JSON
Body Cookies Headers (4) Test Results
Pretty Raw Preview Visualize JSON

```
    "title": "todo3"
}
```

```
{
    "id": 3,
    "title": "todo3"
}
```

Figure 5.12: *POST* **`api/todos`** *API endpoint*

The **`GET api/todos`** endpoint should contain the newly created Todo, as shown in *figure 5.13*.



```
[
    {
        "id": 1,
        "title": "todo1"
    },
    {
        "id": 2,
        "title": "todo2"
    },
    {
        "id": 3,
        "title": "todo3"
    }
]
```

Figure 5.13: *Newly created Todo in GET api/todos*

Now, close the server by hitting *Ctrl* + *c* and start again using the **`aleph dev`** command. You will not see the newly created Todo. This is because we are not saving the data to a file or database. To persist the data, we will create a **`todos.json`** file with the following JSON data in **`db`** folder.

```
// db/todos.json
```

1. `[`
2. `{ “id”: 1, “title”: “todo1” },`

```
{ “id”: 2, “title”: “todo2” }
]
```

The saved todos can then be retrieved using **`readTodos`** utility function created in **`lib/read_todos.ts`** file. We are using **`Deno.readTextFileSync`** function to read the text synchronously and then parse the JSON using **`JSON.parse`**.

```
lib/read_todos.ts
const readTodos = () => JSON.parse(Deno.readTextFileSync(“db/todos.json”));

export default readTodos;
```

Now, we will obtain all saved todos using **`readTodos`** in the **`index.ts`** file. Replace the existing todos code with the following code:

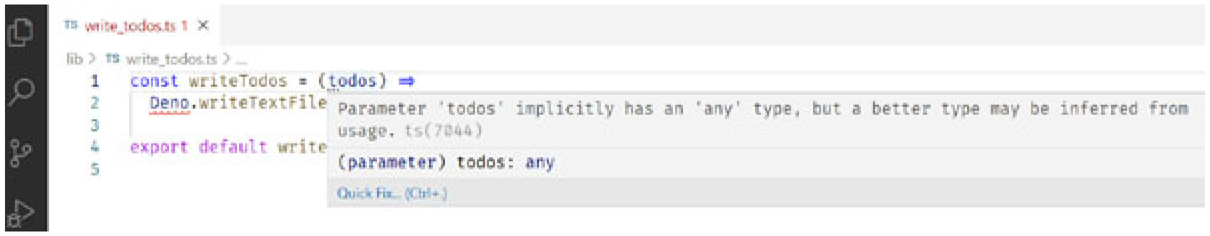
```
// api/todos/index.ts
import type { APIHandler} from “aleph/types.d.ts”;
import readTodos from “../../lib/read_todos.ts”;

export const handler: APIHandler = async ({request, response}) => {
const todos = readTodos();
…
```

We will create another utility function named **`writeTodo`** in the **`lib/write_todo.ts`** file to persist the added Todo by writing it to the **`todos.json`** file.

```
// lib/write_todos.ts
const writeTodos = (todos) =>
Deno.writeTextFileSync(“db/todos.json”, JSON.stringify(todos));
```

```

export default writeTodos;
```

The typescript compiler shows a message for implicit any type for todos parameter as shown in the following figure:



write_todos.ts 1

lib > write_todos.ts > ...

```
const writeTodos = (todos) =>
  Deno.writeTextFile

export default write
```

```
Parameter 'todos' implicitly has an 'any' type, but a better type may be inferred from usage. ts(7044)
(parameter) todos: any
```

Figure 5.14: *TypeScript warning for implicit any type*

Let us fix this by adding an **`ITodo`** interface in the **`interfaces/ITodo.ts`** file.

```
// interfaces/ITodo.ts
export default interface ITodo {
id: number;
title: string;
}
```

We will provide **`ITodo`** as a type for the Todos parameter in **`write_todos.ts`**. The bracket [] after ITodo indicates an ITodo array. The TypeScript warning should be gone now.

```
// utils/write_todos.ts
import ITodo from "../interfaces/ITodo.ts";

const writeTodos = (todos: ITodo[]) =>
…
```

We will use the **`writeTodo`** function and **`ITodo`** interface in the **`index.ts`** file.

```
// api/todos/index.ts
…
```

```
  let todo = todos.find((_todo: ITodo) => _todo.title === title); // Modified code
…
    todos.push(todo);
    writeTodos(todos); // New code
…
```

If you restart the server after adding a Todo, you should see the created Todo persists in the **`todos.json`** file and is included in the **`GET api/todos`** response.

GET (object), PATCH, and DELETE

We will now implement the remaining actions (**`GET`**, **`PUT`**, and **`DELETE`**). All these actions will be performed on the existing Todo. Hence, we will require a unique identifier like the id of the Todo to be passed as a parameter. We will achieve this with the use of the **Dynamic API Route**, which we learned previously in the API section.

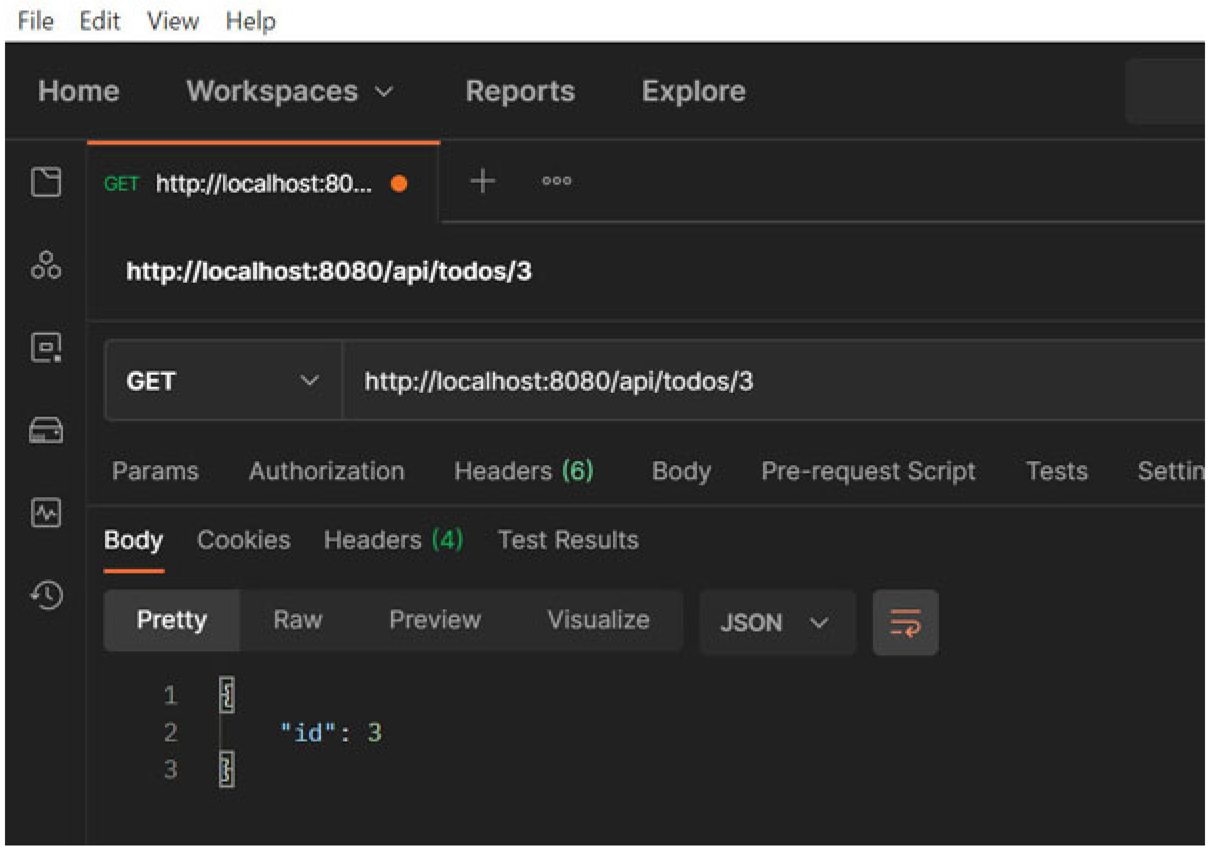
As we learned in the **Routing** section, the bracket syntax is used to create a **Dynamic Route**. We will create the file **`api/todos/[id].ts`** with the following code:

```
// api/todos/[id].ts
import type { APIHandler } from “aleph/types.d.ts”;

export const handler:APIHandler = async ({request, response, router}) => {
const id = parseInt(router.params.id);
response.json({ id: id });
}
```

The function looks for id **`param`** (derived from **`[id].ts`** filename) in the **`router.params`** object. We are using the

parseInt function to convert the id from string to integer. The parsed integer is then returned as a response.

If we hit the endpoint **api/todos/3**, we will get a response as shown in *figure 5.15*.


File Edit View Help
Home Workspaces Reports Explore
GET http://localhost:80...
Params Authorization Headers (6) Body Pre-request Script Tests Settin
Body Cookies Headers (4) Test Results
Pretty Raw Preview Visualize JSON

http://localhost:8080/api/todos/3

GET http://localhost:8080/api/todos/3

```
{
    "id": 3
}
```

***Figure 5.15:** Dynamic API route*

```
1. …
2. import ITodo from “../interfaces/ITodo.ts”;
3. import readTodos from “../../lib/read_todos.ts”;
4.
5. …
6. const todos = readTodos();
7. const id = parseInt(router.params.id);
8. const todo = todos.find((_todo: ITodo) => _todo.id ===
   id);
```

```

if (todo) {
  response.json(todo);
} else {
  response.status = 400;
  response.json({ error: “Todo not found” });
}
…
```

We will use the **`find`** function to get the existing Todo with provided id. If the Todo exists, we will return it, or else we will reply with **`404`** code, and a **`“Todo not found”`** error message.

Just like **`index.ts`**, we will use a switch statement to match **`request.method GET, PATCH, and DELETE`**. The modified code is shown as follows:

```
…

import writeTodos from “../../lib/write_todos.ts”;

…
if (todo) {
  switch (request.method) {
   case “GET”:
    response.json(todo);
    break;

   case “PATCH”:
    let body;
    try {
      body = await request.json();
    } catch (error) {
      response.status = 400;
      response.json({ error: error.message });
      break;
    }

    const title = body.title ? lowerCase(body.title) : null;
    if (!title) {
      response.status = 400
      response.json({
        error: “title not provided”,
      });
      break;
    }

    todos.forEach((_todo: ITodo) => {
      if (_todo.id === todo.id) {
        _todo.title = title;
      }
    });
    writeTodos(todos);
    response.json(todo);
    break;

  case “DELETE”:
    const _todos = todos.filter((_todo: ITodo) => _todo.id !== id);
```

```
      writeTodos(_todos);
      response.json({ message: “Deleted” });
      break;

    default:
      response.status = 405;
      response.json({ message: “Action not allowed” });
      break;
  }
} else {
…
```

For **GET** method, we are simply returning the matched Todo. This **GET** method differs from the one in the **index.ts** file. It provides only a single todo (derived using provided id), whereas the **GET** in **index.ts** provides a list/array of all the todos.

For **DELETE**, we are filtering out the matched Todo from todos and then writing the updated todos to the **todos.json** file using the **writeTodos** function.

For **PATCH**, the initial code is similar to that of the **POST** method from **index.ts** file. We obtain an updated title from the JSON provided by **readBody(“json”)**. Then using **forEach**, we are updating only the matched Todo of todos data. The updated Todos is then saved to the file using the **writeTodos** function, and the updated Todo is replied.

Conclusion

The Todos App API endpoints have been implemented in this chapter. We can now use Postman, or curl to test them. We have also learned the features and basic concepts of Aleph.js.

The next phase will be to implement the UI for Todos App. We will check the unique features offered by Aleph.js for React Development and implement the UI in the upcoming chapter.

CHAPTER 6

Todos App—UI Implementation

Introduction

We have implemented REST APIs for Todos App in the previous chapter. We will implement UI for the same in this chapter. We will be using a Full Stack React framework named **aleph.js** for this purpose. We will also be using the **styled components** library to create UI components and the **axiod** to make network requests.

Structure

In this chapter, we will cover the following topics:

- Styled components
  - Card
  - Text
  - Button
  - Input
  - Checkbox
- Todos components
  - TodoList
  - TodoDetails

Objective

After completion of this chapter, you will be able to create UI for react application using the aleph module. You should also be able to create custom components using styled components and integrate them into the app. You should also be able to make network requests using **`axiod`** module.

Styled components

Styled components is a React library that allows us to create components with css styles having a component-level scope. We will use this library to create components such as **`Button`**, **`Text`**, **`Card`**, and so on. Switch over to the **`todos-app`** folder from a terminal that was initialized in the previous chapter. To install styled-components, we will use the following **`trex`** command:

```
trex -c styled=”https://esm.sh/styled-components@5.3.1”
```

Once installed, we will create various components in **`StyledComponents.tsx`** file. The code for each component will be as follows:

```
import styled, { css } from “styled”

export const Card = styled.div`
    display: flex;
    flex-direction: row;
    background: #ddd;
    border-radius: 5px;
    margin: 10px;
    justify-content: space-between;
    align-items: center;
    max-width: 600px;
    margin: “auto”;
`
```

```

export const Text = styled.span`
    font-size: 20px;
    font-weight: bold;
    flex: 1;
    margin-left: 10px;
     ${props => props.completed && css`
        text-decoration: line-through;
        color: grey
     `}
`
export const Button = styled.button`
  /* Adapt the colors based on primary prop */
  background: ${props => props.primary ? “palevioletred” : “white”};
     color: ${props => props.primary ? “white” : “palevioletred”};

  font-size: 1em;
  margin: 1em;
  padding: 0.25em 1em;
  border: 2px solid palevioletred;
  border-radius: 3px;
  cursor: pointer;
  ${props => props.disabled && css`
      background: lightgrey;
      cursor: not-allowed;
      border: 2px solid lightgrey;
    `
    }
`;

export const Input = styled.input`
    flex:1;
    padding: 10px;
    margin: 10px;
    border: none;
    background: #ddd;
    font-size: 20px;
    font-weight: bold;
     ${props => props.completed && css`
        text-decoration: line-through;
        color: grey
     `}
`

export const Checkbox = styled.input`
    type: “checkbox”;
    color: red;
    margin-left: 10px;
`

export const Header1 = styled.h1`

`
```

In the preceding code, we are styling various components such as **`Card`**, **`Button`**, **`Text`**, and so on using styled and css from **`styled`** components. Each component is then exported and used across the project. We are creating our own UI library using this approach.

We are passing a function to some components. These functions are used to set values based on props. We are using the **`completed`** prop in **`Text`** and **`Input`** components to adapt color and text decoration accordingly. We are using the **`primary`** flag in the **`Button`** component to adapt the color as per the prop.

Creating Todos components

We will now use the custom-styled components in **`TodoDetails`** and **`TodoList`** components. We will start by adding the **`TodoList`** component in **`pages/index.tsx`** file, which is the entry point of our project as follows:

```
import React from ‘react’
import TodoList from “../components/TodoList.tsx”

export default function Home() {
  return <TodoList />
}
```

We will check the details of the **`TodoList`** component in the next section.

TodoList

This component will be used to add new Todo and list existing Todos. We will be using custom components from **`StyledComponents.tsx`** file for this purpose. The API calls will be made using **`axiod`** module. The code for this file is as follows:

```
// components/TodoList.tsx

import React from ‘react’
import { useDeno } from “https://deno.land/x/aleph/framework/react/mod.ts”;
import axiod from “axiod”;

import TodoDetails from “./TodoDetails.tsx”;
import { Button, Card, Input, Header1 } from “./StyledComponents.tsx”;

import readTodos from “../lib/read_todos.ts”;

export default function Page() {
  const _todos = useDeno(async () => {
    return await readTodos();
  }, { revalidate: 1 });
  const [todos, setTodos] = React.useState(_todos);
  const [todoInput, setTodoInput] = React. useState(“”);

  const getTodos = () => {
    axiod.get(“/api/todos”).then((res) => {
      console.log(res);
      setTodos(res.data);
    }).catch((err) => {
      console.log(err);
    });
  };

  const addTodo = () => {
    if (!todoInput) {
      alert(“Please enter a todo text”);
      return;
    }
    axiod.post(“/api/todos”, { title: todoInput }).then((res) => {
      console.log(res);
      getTodos();
      alert(“Todo added successfully”);
    }).catch((err) => {
      console.log(err);
      alert(“Error adding todo”);
    });
  };
  return (
    <>
      <Header1>Todos</Header1>
      <Card>
        <Input
          placeholder=”Add Todo”
          value={todoInput}
          onChange={(e) => setTodoInput(e.target. value)}
        />
        <Button primary onClick={addTodo}>+</Button>
      </Card>
      {todos.map((todo: any) => <TodoDetails key={todo.id} getTodos={getTodos} todo={todo} />)}
    </>
  );
```

```
56. }
```

In the preceding code, we are importing the custom components, **axiod** module and **readTodos** function. We are also importing the **useDeno** function that allows Deno runtime usage in the component. We are fetching the Todos from JSON file in it. The state **todos** is used to store the list of Todos, whereas **todoInput** is used to store add Todo text box input.

We are using **getTodos** function to fetch Todos by making an API call using **axiod** module. The fetched Todos are then stored in **todos** state. Similarly, the **addTodo** function is used to make **POST** requests to add new Todo in the JSON file. We are passing **todoInput** as **title** in the request body. If the Todo is added successfully, we fetch the Todos using **getTodos** function.

The styled-components **Header1**, **Card**, **Input**, and **Button** are used for UI creation. **TodoDetails** component is used to render each Todo detail. We will check its details in the next section.

Figure 6.1 shows the **TodoList** component with Add Todo Input.

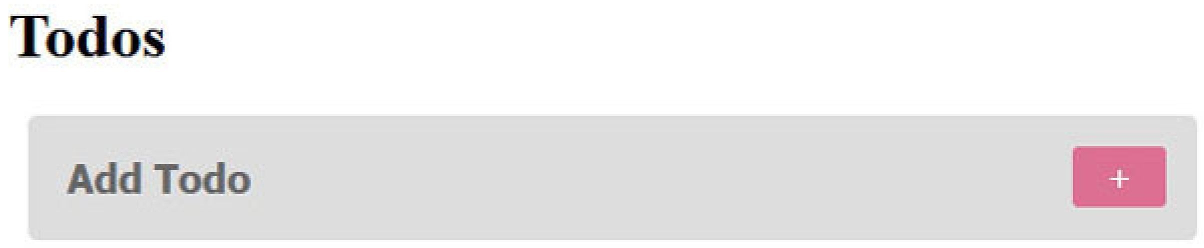


Figure 6.1: *TodoList component*

TodoDetails

We will be using this component to render each added **todo** element. We will provide two actions, namely, **Save** and **Delete**. We will also use a **checkbox** element to mark the Todo as completed. The code for this component is written in **`TodoDetails.tsx`** file as follows:

```
// components/TodoDetails.tsx

import React from “react”;
import { Button, Card, Checkbox, Input, Text } from “./StyledComponents.tsx”;
import axiod from “axiod”;

interface ITodo {
  id: number;
  title: string;
  completed: boolean;
}

interface IProps {
  todo: ITodo;
  getTodos: () => void;
}

function TodoDetails(props: IProps) {
  const { todo, getTodos } = props;
  const [isCompleted, setIsCompleted] = React.useState(todo.completed);
  const [todoText, setTodoText] = React.useState(todo.title);

  const editTodo = () => {
    if (!todoText) {
      alert(“Please enter a todo text”);
      return;
    }
    axiod.patch(`/api/todos/${todo.id}`, {
      title: todoText,
      completed: isCompleted,
    }).then((res) => {
      console.log(res);
      alert(“Todo updated successfully”);
    }).catch((err) => {
      console.log(err);
      alert(“Error updating todo”);
      setTodoText(todo.title);
    });
  };

  const deleteTodo = () => {
    axiod.delete(`/api/todos/${todo.id}`, { title: todoText }).then((res) => {
      console.log(res);
      alert(“Todo deleted successfully”);
      getTodos();
    }).catch((err) => {
      console.log(err);
      alert(“Error deleting todo”);
    });
  };

  return (
    <Card>
      <Checkbox
        type=”checkbox”
        defaultChecked={isCompleted}
        onChange={() => setIsCompleted(!isCompleted)}
      />
      <Input
        completed={isCompleted}
        value={todoText}
        onChange={(e) => setTodoText(e.target.value)}
        disabled={isCompleted}
      />
      <Button primary onClick={editTodo}>Save</Button>
      <Button onClick={deleteTodo}>Delete</Button>
    </Card>
  );
}

export default TodoDetails;


```

In the preceding code, we are importing custom-styled components **`Button, Card, Checkbox, Input, Text,`** and **`axiod`**. We then defined two interfaces **`ITodo`** (Todo details) with **`id, title,`** and **`completed`** fields and **`IProps`** (props passed by parent component) with **`todo`** and **`getTodos`** fields. We have also defined two states **`isCompleted`** (to mark Todo as completed) with **`todo.completed`** as the initial value and **`todoText`** (to store user input) with **`todo.title`** as the initial value.

We have defined two actions **`editTodo`** and **`deleteTodo`**, for updating the Todo and deleting the Todo, respectively. In **`editTodo`**, we check if the edited **`todoText`** is empty or not. We show the message if it is empty; else, we make a **`PATCH`** request using **`axiod`** passing **`todoText`** as **`title`** and **`isCompleted`** as **`completed`** flag. In **`deleteTodo`** action, we are passing **`todo.id`** as a param and fetching the Todos list if the delete request is successful.

Figure 6.2 shows the **`TodoDetails`** component with Save and Delete actions along with a checkbox.

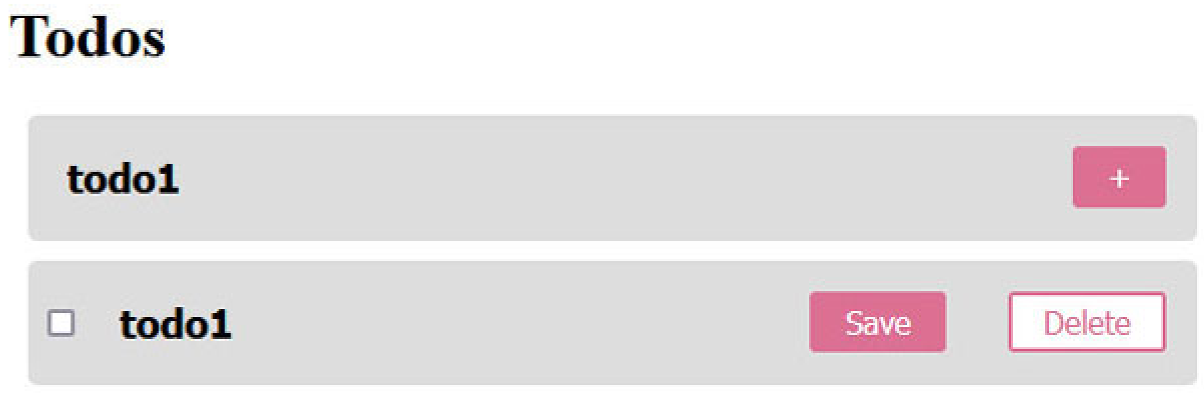


Figure 6.2: *TodoDetails component*

For the JSX part, we are using the **`Card`** element as a container. The **`Checkbox`** element has **`isCompleted`** as default value and the **`onChange`** function will invoke **`setIsCompleted`** function with negated **`isCompleted (!isCompleted)`**. The **`Input`** element has **`completed`** prop set to **`isCompleted`** value. This is done to change the styling of the element according to the flag as shown in the **`StyledComponents.tsx`** file. The element is bound with **`todoText`** using **`value`** and **`onChange`** attributes. The **`disabled`** flag is also set to **`isCompleted`** similar to **`completed`** flag. In the end, we are **`Button`** element with **`onClick`** attribute to **Edit** or **Delete** the Todo by invoking the respective function.

Figure 6.3 shows the **`TodoDetails`** component with completed status.

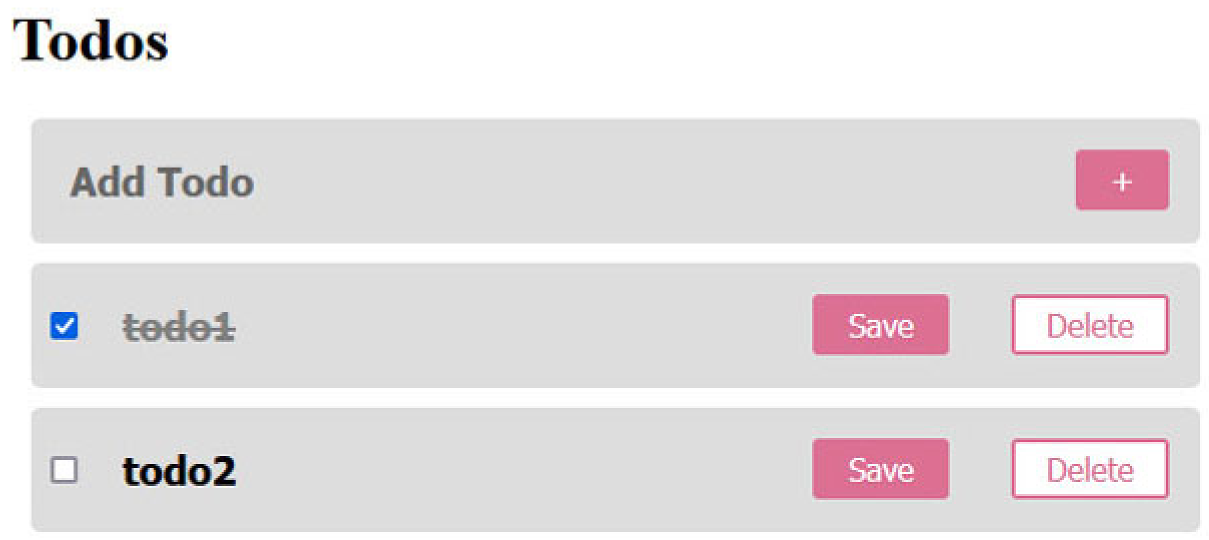


Figure 6.3: *TodoDetails component with completed status*

Conclusion

We have created a UI for the Full Stack React application in this chapter. We used the Aleph framework for this purpose. We also used the Styled components module to create custom components. Axiod was used to make API calls.

This knowledge will be useful for creating a Full Stack App having API endpoints and UI using aleph. Our Todos App is completed, and we will create a different project in the upcoming chapters. We will create a GraphQL API server using **`oak`** and **`oak-graphql`** modules in the next chapter, and then we will create UI for a svelte app using snel module in the subsequent chapter.

CHAPTER 7
Introduction to GraphQL, Oak, and Oak-GraphQL

Introduction

We are going to learn the core **Graph Query Language** (**GraphQL**) concepts along with its features in this chapter. Once we cover the basics, we shall create an API server (GraphQL) for the *Phonebook* App using these concepts in the upcoming chapter. We will also introduce the *Oak Web* framework and *Oak GraphQL*. Both will be used for GraphQL API server implementation in the upcoming chapter.

Structure

In this chapter, we will cover the following topics:

- GraphQL intro
    - Features
        - Precise fetching
        - Single API endpoint
        - Validation and type checking
        - Fragments
        - Auto-generated API docs
    - Core concepts
        - Schema
            - Type
            - Input

            - Query
            - Mutation
            - Relationship
        - Resolver
- Introducing Oak framework
- Introducing Oak-GraphQL

Objective

After completion of this chapter, you will be able to understand and apply GraphQL core concepts such as query, mutation, resolvers, type definitions, and so on. You should also be able to use the Oak framework to create Deno Web servers and the Oak-GraphQL module to create GraphQL servers.

Introducing GraphQL

GraphQL is an open-source query language for APIs developed by Facebook and released in 2015. It allows the client to control the resolved data structure as opposed to a fixed data structure in REST. This way, the client can **query** only for the exact necessary data. It also allows **mutations (**writing or changing data) and **subscriptions** (subscribing to data updates). It is used by many large organizations, such as Facebook, GitHub, Coursera, and Pinterest, to name a few.

Features

GraphQL allows clients to *query or mutate* data across multiple resources in a single request. It creates a single entry point or API endpoint POST **/graphql** to query or mutate the data. GraphQL also provides resolved data in the same format as that of the requested query. Other GraphQL features are explained in detail as follows.

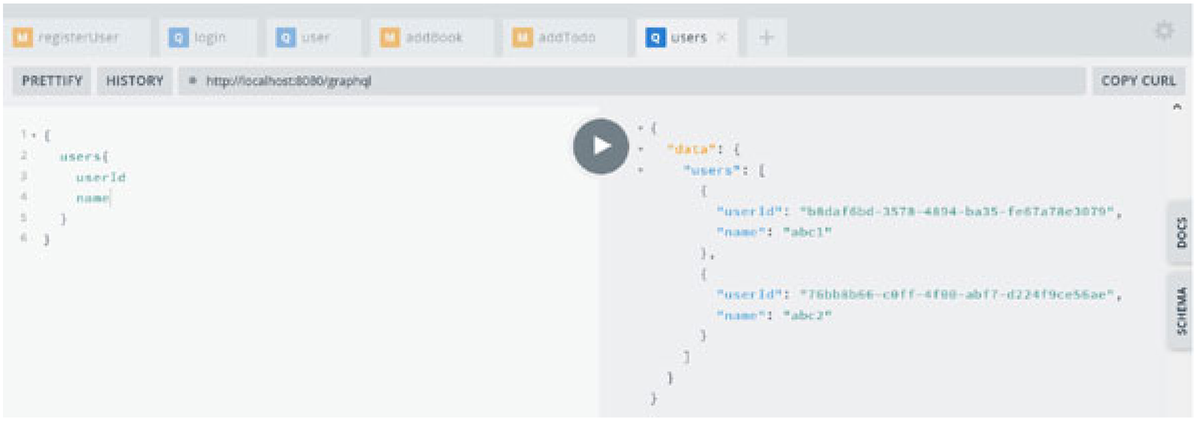
Precise fetching

By allowing the client to fetch only required data, GraphQL solves the data-related issues where the client gets more than required data (*over fetching*) or less than required data (*under fetching*).

Single API endpoint

Unlike REST, GraphQL will always have a single Endpoint **/graphql (POST)**, which can be used by the client to fetch any data defined in the schema as GraphQL resolves the request using the *query* passed in the request body.

Figure 7.1 fetches **`users`** data using **/graphql** endpoint by providing the **`users`** query in the body as follows:

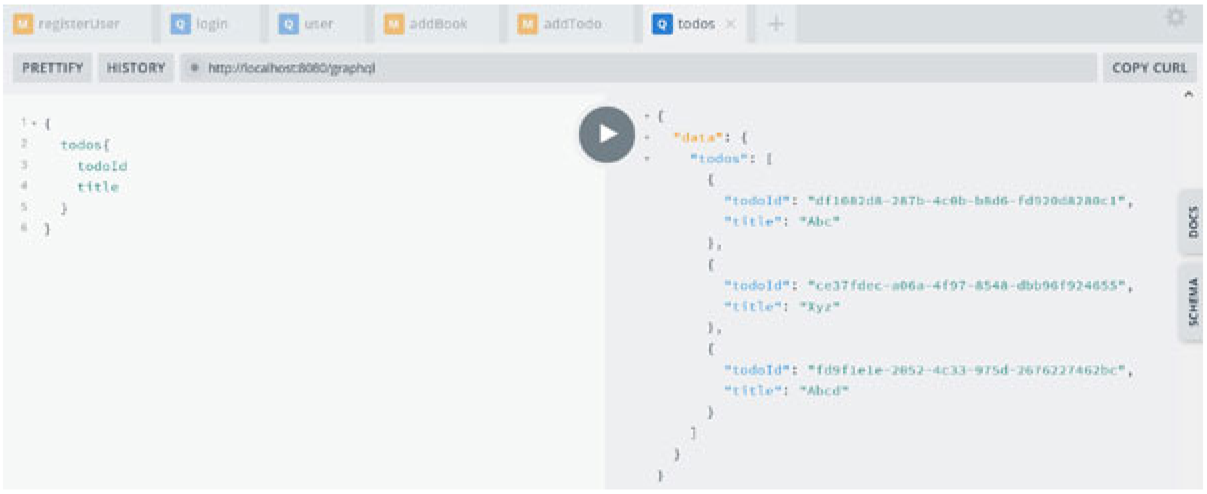


```
{
  users{
    userId
    name
  }
}
```

```
{
  "data": {
    "users": [
      {
        "userId": "b8daf6bd-3578-4894-ba35-fe67a78e3079",
        "name": "abc1"
      },
      {
        "userId": "76bb8b66-c0ff-4f80-abf7-d224f9ce56ae",
        "name": "abc2"
      }
    ]
  }
}
```

Figure 7.1: *Fetch users*

Figure 7.2 fetches *Todos data* using the same endpoint **/graphql** by providing **`todos`** query in the body as follows:



```
{
  todos{
    todoId
    title
  }
}
```

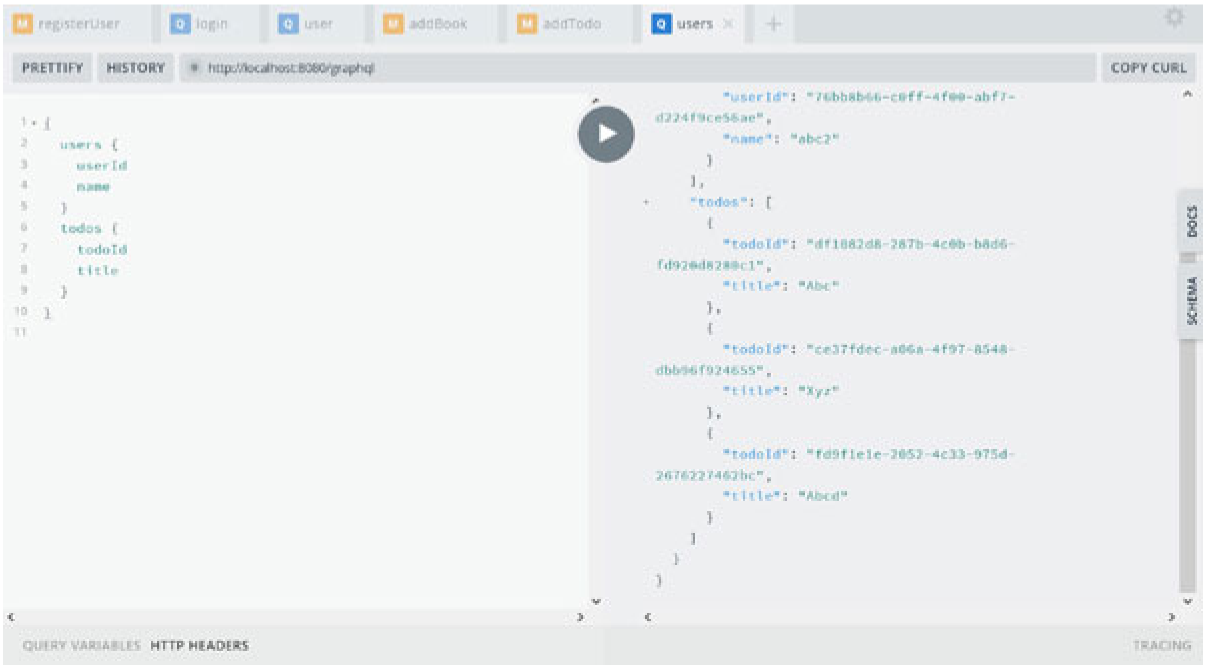
```
{
  "data": {
    "todos": [
      {
        "todoId": "df1082d8-287b-4c0b-b8d6-fd920d8280c1",
        "title": "Abc"
      },
      {
        "todoId": "ce37fdec-a06a-4f97-8548-dbb96f924655",
        "title": "Xyz"
      },
      {
        "todoId": "fd9f1e1e-2052-4c33-975d-2676227462bc",
        "title": "Abcd"
      }
    ]
  }
}
```

Figure 7.2: *Fetch Todos*

As shown in *figures 7.1* and *7.2*, GraphQL allows fetching **`users`** and **`todos`** from a single API endpoint using **query** operation**.**

We can even fetch **users** and **todos** in a single request. This is a very powerful feature of GraphQL that lets the client get all the required data in a single request, which is extremely useful for devices with slower networks.

Figure 7.3 fetches both **`users`** and **`todos`** in a single request as follows:



```
{
  users {
    userId
    name
  }
  todos {
    todoId
    title
  }
}
```

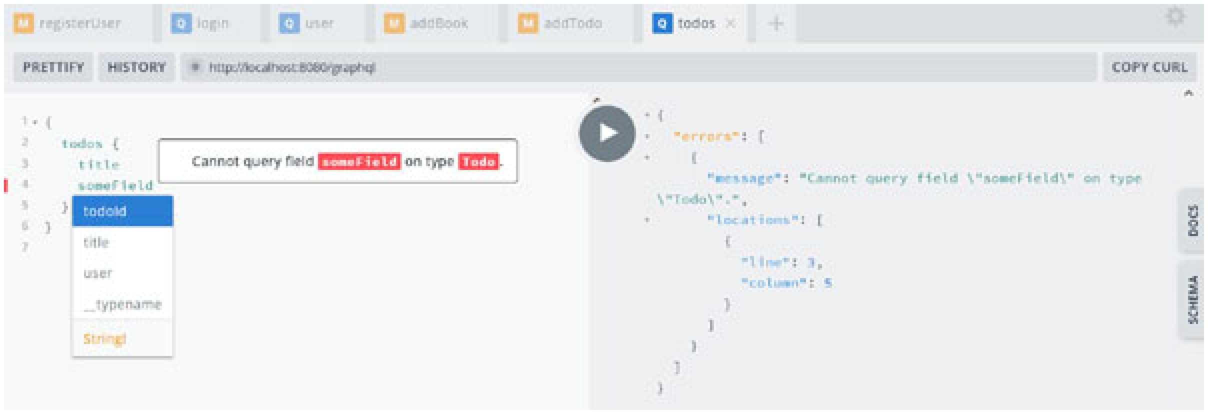
```
        "userId": "76bb8b66-c0ff-4f00-abf7-
d224f9ce56ae",
        "name": "abc2"
      }
    ],
    "todos": [
      {
        "todoId": "df1082d8-287b-4c0b-b8d6-
fd920d8280c1",
        "title": "Abc"
      },
      {
        "todoId": "ce37fdec-a06a-4f97-8548-
dbb96f924655",
        "title": "Xyz"
      },
      {
        "todoId": "fd9f1e1e-2052-4c33-975d-
2676227462bc",
        "title": "Abcd"
      }
    ]
  }
}
```

Figure 7.3: *Fetch Users and Todos*

Validation and type checking

GraphQL uses **a type system** to validate the query. The client can check which fields are available (with field type) and whether the query is valid or not before making the request.

Figure 7.4 demonstrates validation and type-checking concepts for **`todos`** where **`someField`** field is not defined in the schema as follows:



```
{
  todos {
    title
    someField
  }
}
```

Cannot query field someField on type Todo.

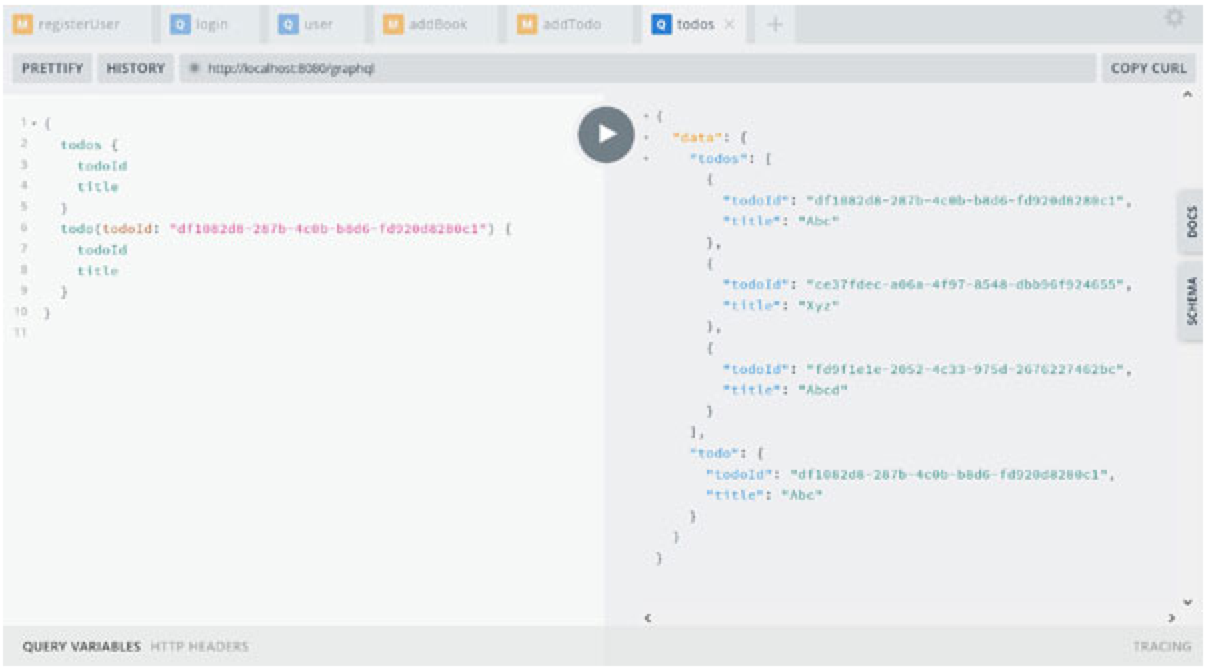
```
{
  "errors": [
    {
      "message": "Cannot query field \"someField\" on type
\"Todo\".",
      "locations": [
        {
          "line": 3,
          "column": 5
        }
      ]
    }
  ]
}
```

Figure 7.4: *Validation*

Fragments

GraphQL allows writing a reusable code *fragment* that can be used across multiple queries. We can create a fragment that replaces the repeated code unit and makes the queries less cluttered and more readable.

Figure 7.5 shows code units repeated for **`todos`** (fetch all Todos) and **`todo`** (fetch single Todo by id) queries as follows:



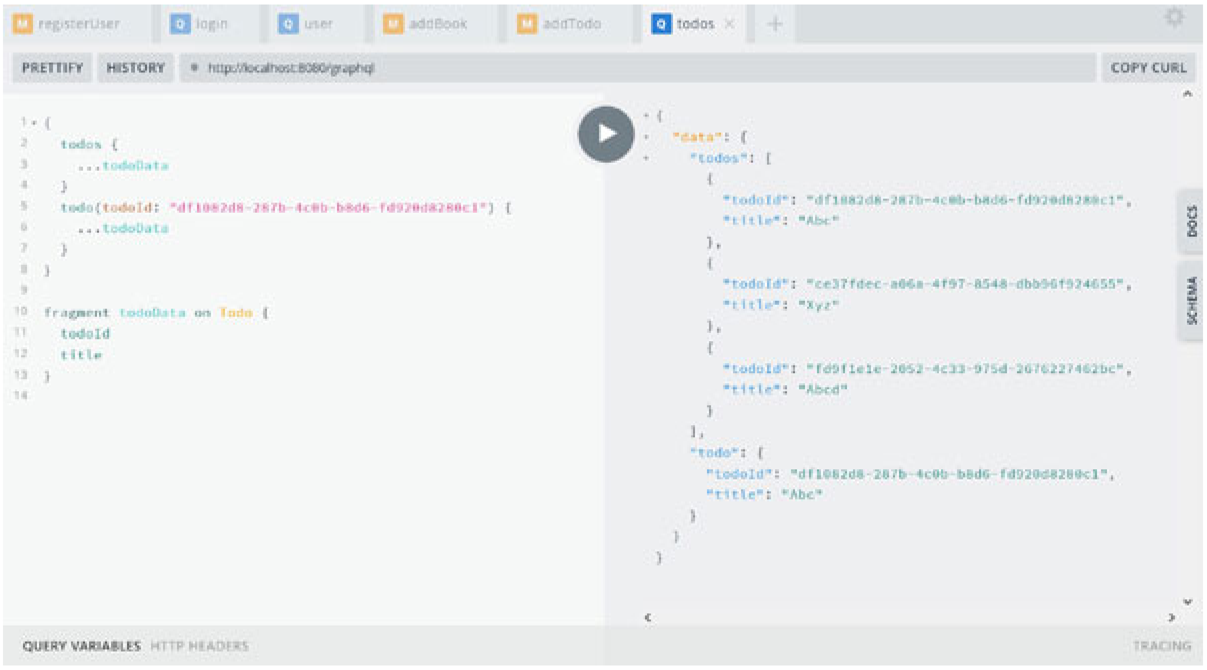
```
{
  todos {
    todoId
    title
  }
  todo(todoId: "df1082d8-287b-4c0b-b8d6-fd920d8280c1") {
    todoId
    title
  }
}
```

```
{
  "data": {
    "todos": [
      {
        "todoId": "df1082d8-287b-4c0b-b8d6-fd920d8280c1",
        "title": "Abc"
      },
      {
        "todoId": "ce37fdec-a06a-4f97-8548-dbb56f524655",
        "title": "Xyz"
      },
      {
        "todoId": "fd9f1e1e-2052-4c33-975d-2676227462bc",
        "title": "Abcd"
      }
    ],
    "todo": {
      "todoId": "df1082d8-287b-4c0b-b8d6-fd920d8280c1",
      "title": "Abc"
    }
  }
}
```

Figure 7.5: *Repeated code unit*

We will move the repeated code unit in the **`todoData`** fragment and use it with the *spread operator* (…) in both queries.

Figure 7.6 demonstrates the use of **`fragment`** for fetching **`todoData`** and **`userData`** as follows:



```
{
  todos {
    ...todoData
  }
  todo(todoId: "df1082d8-287b-4c0b-b8d6-fd920d8280c1") {
    ...todoData
  }
}

fragment todoData on Todo {
  todoId
  title
}
```

```
{
  "data": {
    "todos": [
      {
        "todoId": "df1082d8-287b-4c0b-b8d6-fd920d8280c1",
        "title": "Abc"
      },
      {
        "todoId": "ce37fdec-a06a-4f97-8548-dbb96f924655",
        "title": "Xyz"
      },
      {
        "todoId": "fd9f1e1e-2052-4c33-975d-2676227462bc",
        "title": "Abcd"
      }
    ],
    "todo": {
      "todoId": "df1082d8-287b-4c0b-b8d6-fd920d8280c1",
      "title": "Abc"
    }
  }
}
```

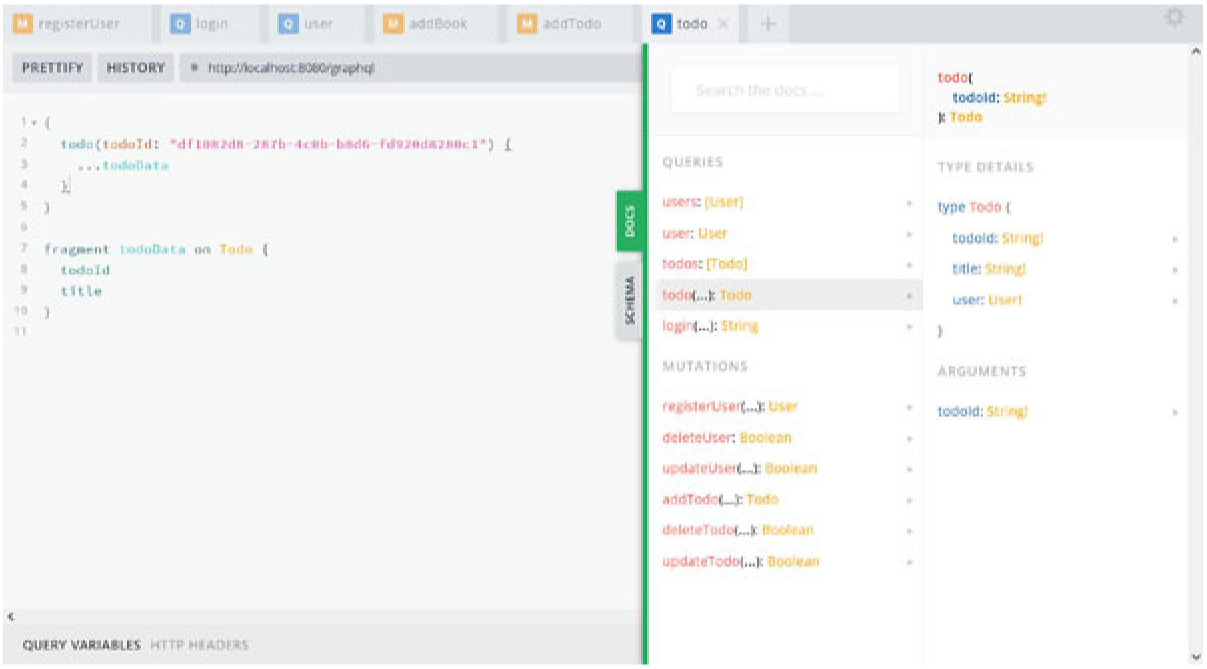
Figure 7.6: *Fragments*

Fragments are very useful in complex queries used in multiple places with many fields.

Autogenerated API documentation

GraphQL generates API documentation using the schema for *queries* and *mutations*.

Figure 7.7 shows autogenerated API documentation for **`todo`** using schema.



```
{
  todo(todoId: "df1082d8-287b-4c0b-b8d6-fd920d8280c1") {
    ...todoData
  }
}

fragment todoData on Todo {
  todoId
  title
}
```

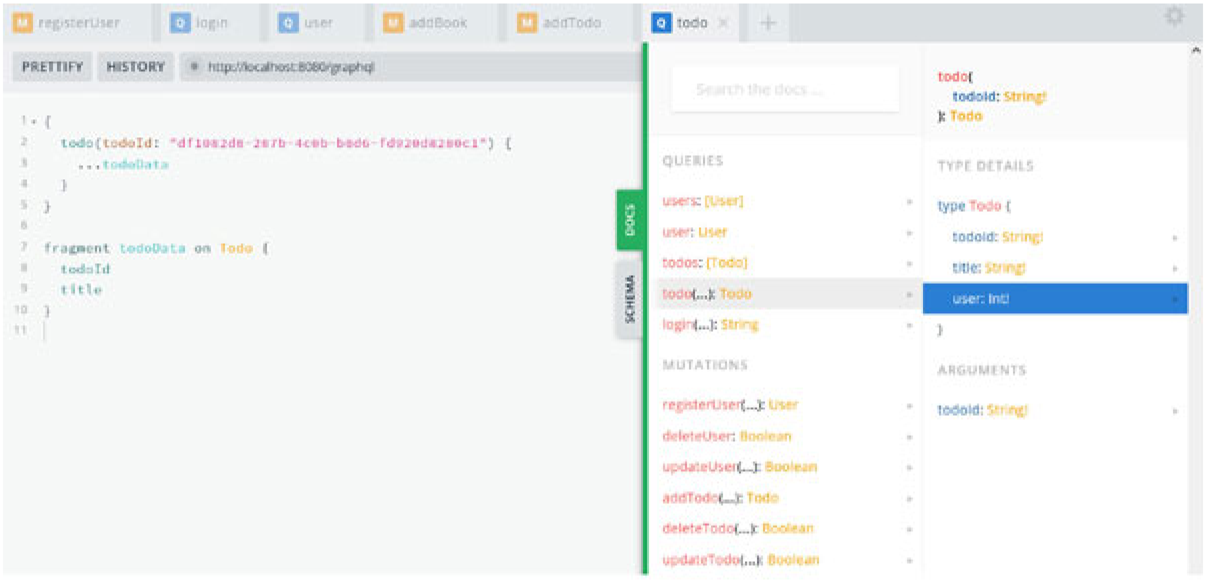
```
QUERIES
users: [User]
user: User
todos: [Todo]
todo(...): Todo
login(...): String

MUTATIONS
registerUser(...): User
deleteUser: Boolean
updateUser(...): Boolean
addTodo(...): Todo
deleteTodo(...): Boolean
updateTodo(...): Boolean
```

```
todo(
  todoId: String!
): Todo

TYPE DETAILS
type Todo {
  todoId: String!
  title: String!
  user: User!
}

ARGUMENTS
todoId: String!
```

Figure 7.7: *API docs*

As the documentation is generated according to the schema, every change in the schema is synced and reflected in the docs.

If we change the **`user`** type from **`User`** to **`Int`**, docs will reflect it, as shown in *figure 7.8*.



```
{
  todo(todoId: "df1082d8-287b-4c0b-b8d6-fd920d8280c1") {
    ...todoData
  }
}

fragment todoData on Todo {
  todoId
  title
}
```

```
QUERIES
users: [User]
user: User
todos: [Todo]
todo(...): Todo
login(...): String

MUTATIONS
registerUser(...): User
deleteUser: Boolean
updateUser(...): Boolean
addTodo(...): Todo
deleteTodo(...): Boolean
updateTodo(...): Boolean
```

```
todo(
  todoId: String!
): Todo

TYPE DETAILS
type Todo {
  todoId: String!
  title: String!
  user: Int!
}

ARGUMENTS
todoId: String!
```

Figure 7.8: *Synced docs*

Core concepts

GraphQL provides a *Schema* for defining the data model, *a Resolver* that *resolves* the *request*, along with actions like *query* for reading data, *mutations* for writing data, and *subscription* for subscribing to real-time data. We will cover these concepts in the following section.

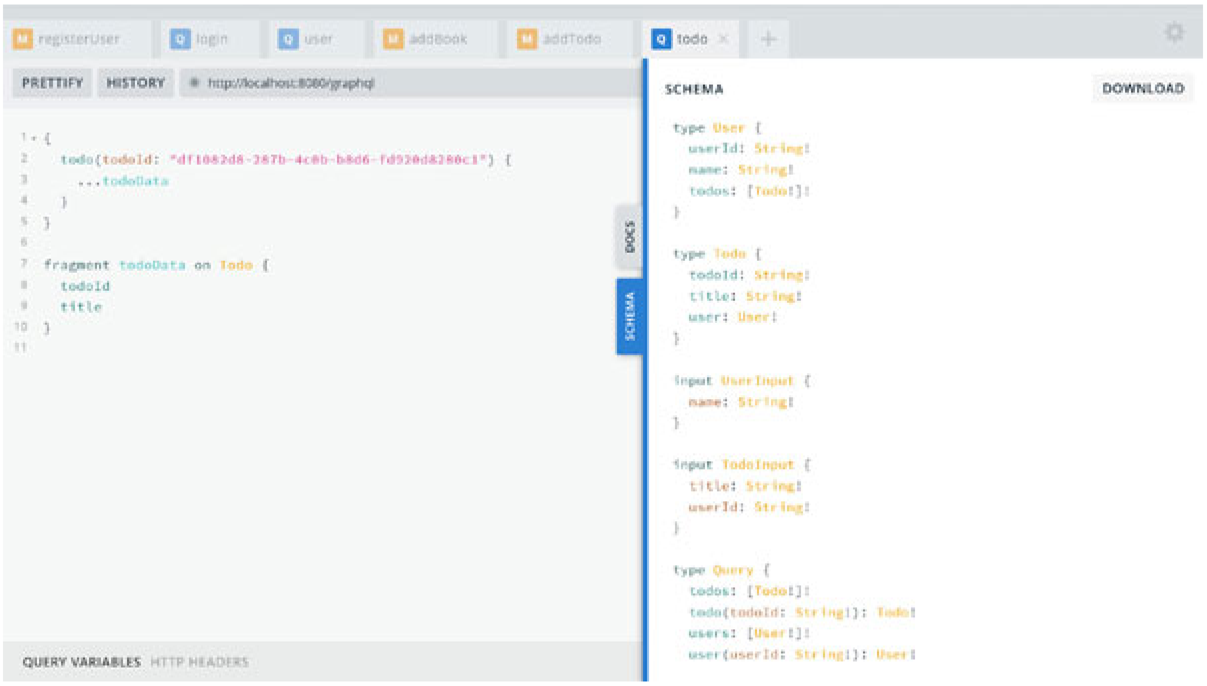
Schema

A Schema (*a type system*) is a contract between client and server that defines the shape of the data. The schema defines *types* that describe an entity or field and the relationship between types. It lets the client discover available *data fields*, *queries***,** and *mutations*. The client uses this schema to define the structure of resolved data.

GraphQL uses **Schema Definition Language** (**SDL**) to define the schema. A schema for a sample Todos app can be defined as follows:

```
type User {
  userId: String!
  name: String! # Scalar Type
  todos: [Todo!]! # Object Type
}

type Todo {
  todoId: String!
  title: String!
  user: User!
}

input UserInput {
  name: String!
}

input TodoInput {
  title: String!
  userId: String!
}

type Query {
  todos: [Todo!]!
  todo(todoId: String!): Todo!
  users: [User!]!
  user(userId: String!): User!
}

type Mutation {
  addTodo(todo: TodoInput!): Todo
  addUser(user: UserInput!): User
}
```

This schema can also be viewed via *GraphQL playground*, as shown in *figure 7.9*.



```
{
  todo(todoId: "df1082d8-287b-4c0b-b8d6-fd920d8280c1") {
    ...todoData
  }
}

fragment todoData on Todo {
  todoId
  title
}
```

```
type User {
  userId: String!
  name: String!
  todos: [Todo!]!
}

type Todo {
  todoId: String!
  title: String!
  user: User!
}

input UserInput {
  name: String!
}

input TodoInput {
  title: String!
  userId: String!
}

type Query {
  todos: [Todo!]!
  todo(todoId: String!): Todo!
  users: [User!]!
  user(userId: String!): User!
```

Figure 7.9: *GraphQL schema*

Let us explain various keywords (**`type`** and **`input`**) used in the preceding schema:

Type

Type is used to define the shape of the resolved query. The schema shown in *figure 7.9* defines two Object types **`User`** and **`Todo`**. **`User`** type defines fields **`userId`**, **`name`**, and **`todos`** (an array of **`Todo`** object type). **`Todo`** type defines fields **`todoId`**, **`title`**, and **`user`** (of type **`User`**). Both object types are used by the **`Query`** type while returning the corresponding data.

Input

Input is used to define the shape of client input during *mutation*. It defines two *input types*, **`UserInput`** and **`TodoInput`**. **`UserInput`** accepts a **`name`** argument, and **`TodoInput`** accepts **`title`** and **`userId`** arguments. Both Input types are used for creating or updating the corresponding entity by the *mutation* type.

Query

Defines entry points for **`read`** operations to be executed by the client. Four fields **`todos`**, **`todo`**, **`users`**, and **`user`** are defined in the query. **`todos`** and **`users`** fields accept no arguments and return array corresponding types. **`todo`** and **`user`** fields accept the **`id`** of the entity and return the entity of corresponding types.

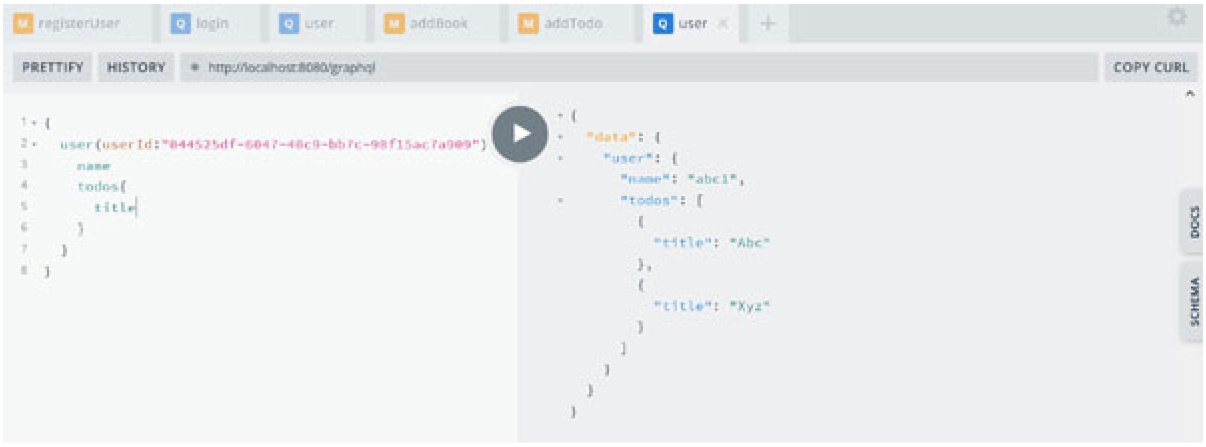
Mutation

Defines entry points for **write** operations to be executed by the client. **`addTodo`** and **`addUser`** fields are defined in this type. **`addTodo`** accepts arguments of type **`TodoInput`** and returns data of type **`Todo`,** whereas **`addUser`** accepts arguments of type **`UserInput`** and returns data of type **`User`**.

Relationship

The Todos schema defines the relationship of type *one-to-many* between **`User`** and **`Todo`** types. It means every *Todo* has *one* user, and every *user* has *many* Todos**.**

As shown in *figure 7.10*, GraphQL gives flexibility to the client to fetch all user-related Todos in a single query.



```
{
  user(userId:"044525df-6047-48c9-bb7c-98f15ac7a909")
    name
    todos{
      title
    }
  }
}
```

```
{
  "data": {
    "user": {
      "name": "abc1",
      "todos": [
        {
          "title": "Abc"
        },
        {
          "title": "Xyz"
        }
      ]
    }
  }
}
```

Figure 7.10: *Relationships*

Resolver

A resolver is a function used to *resolve* a *query* or *mutation* by populating data for a single field in a schema. The following code demonstrates the use of *resolvers* for a previously defined schema**:**

```
export const resolvers = {
  Query: {
    todos: (parent: any, args: any, context: any, info: any) => {
      if (args.userId) return getTodosByUserId(args.userId);
      return getTodos();
    },
    todo: async (parent: any, args: any, context: any, info: any) => {
      return getTodo(args._id);
    },
    users: (parent: any, args: any, context: any, info: any) => {
      return getUsers();
    },
    user: async (parent: any, args: any, context: any, info: any) => {
      return getUser(args._id);
    },
  },
  Mutation: {
    addTodo: async (parent: any, args: any, context: any, info: any) => {
      const { title, userId } = args.todo;
      return createTodo(title, userId);
    },
    addUser: async (parent: any, args: any, context: any, info: any) => {
      const { name } = args.user;
      const user = createUser(name);
      return user;
    },
  },
  Todo: {
    title: (parent: any, args: any, context: any, info: any) => {
      return parent.title;
    },
    user: async (parent: any, args: any, context: any, info: any) => {
      const user = await info.rootValue.Query.user(
        parent,
        { _id: parent.userId },
        context,
        info
      );
      return user;
    },
  },
  User: {
    name: (parent: any, args: any, context: any, info: any) => {
      return parent.name;
    },
    todos: async (parent: any, args: any, context: any,
info: any) => {
      const todos = await resolvers.Query.todos(
        parent,
        { userId: parent._id },
        context,
        info
      );
      return todos;
    },
  },
};

```

Let us see what is happening in the preceding code. We define **`Query`**, **`Mutation`**, **`Todo`**, and **`User`** objects. **`Query`** object contains **`todos`** (list), **`todo`**, **`users`**, and **`user`** objects. Similarly, **`Mutation`** contains **`addTodo`** and **`addUser`** objects, each with the same signature. The other objects **`User`** and **`Todo`** are required for resolving internal fields (title and so on) of the respective type. The data can then be fetched from various sources. Each object mentioned previously accepts optional parameters: parent**, `args`, `context`**, and **`info`**.

The **`parent`** is the return value for the field's parent, and **`args`** contains all the provided GraphQL arguments, a **`context`** is an object whose content is available across all resolvers, and **info** contains information such as field name, path, and so on.

Introducing Oak framework

Oak is a Web framework for Deno, inspired by the *koa* framework of node.js. It provides middleware functions that are invoked in between the request and the response. Oak provides the **`use()`** method to use the middleware function. The following code demonstrates the **`Hello World`** server in oak:

```
import { Application } from "https://deno.land/x/oak@v7.7.0/mod.ts";

const app = new Application();

app.use((ctx) => {
  ctx.response.body = "Hello World!";
});

await app.listen({ port: 8000 });
```

In the preceding code, the **`Application`** instance **`app`** invokes each middleware with *context*, which contains necessary info such as **`app`**, **`cookies`**, **`request`**, **`response`**, **`state`**, **`assert()`**, and **`throw()`**. Finally, the server starts listening on port **`8000`** for incoming requests using **`listen()`** method.

Introducing Oak-GraphQL middleware

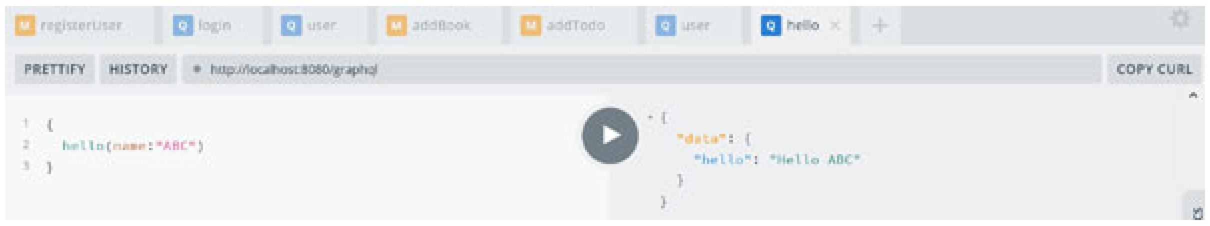
Oak-GraphQL module allows us to create a GraphQL API on top of the oak server. We will be using Oak-GraphQL middleware for GraphQL API implementation. A simple Hello World GraphQL API can be set up as follows:

```
// oak_graphql_server.ts

import {
  Application,
  Router,
  RouterContext,
} from “https://deno.land/x/oak@v9.0.0/mod.ts”;
import {
  applyGraphQL,
  gql,
  GQLError,
} from “https://deno.land/x/oak_graphql/mod.ts”;

const app = new Application();
const types = gql`
  type Query {
    hello(name: String): String
  }
`;

const resolvers = {
  Query: {
      hello: (parent: any, { name }: any, context: any, info: any) => {
      if (!name) throw new GQLError(“Name not provided”);
      return `Hello ${name}`;
    },
  },
};

const GraphQLService = await applyGraphQL<Router>({
  Router,  // Same as Router: Router,
  typeDefs: types,
  resolvers: resolvers,
});

app.use(GraphQLService.routes(), GraphQLService.allowedMethods());

console.log("Server listening on http://localhost:8080/graphql");
await app.listen({ port: 8080 });
```

Preceding code imports required methods and classes from **`oak`** (Application, Router, and RouterContext) and **`oak-graphql`** (applyGraphQL, gql, and GQLError). After creating an **`Application`** instance, type definitions are defined using **`gql`** along with **resolvers** for the type definitions.

A **`GraphQLService`** is then created using the **`applyGraphQL`** function that accepts an object with a **`Router`**, **`typeDefs`**, and **`resolvers`**. Middlewares **`GraphQLService.routes()`**, **`GraphQLService.allowedMethods()`** are then added to the **`app`** using the **`use()`** method. Finally, the server starts and listens to incoming requests using the **`listen()`** method on port 8080.

If we run the preceding code with the command **`deno run --allow-net oak_graphql_server.ts`** and open URL **http://localhost:8080/graphql** with query **`{ hello(name: "ABC") }`** we should see the results shown in *figure 7.11* as follows:



```
{
  hello(name:"ABC")
}
```

```
{
  "data": {
    "hello": "Hello ABC"
  }
}
```

Figure 7.11: *GraphQL API Hello World*

Conclusion

We have covered the basic features and concepts of GraphQL, such as schema, resolver along with type, query, and mutation. We also introduced the oak framework used to create a Web server along with the Oak-GraphQL module used to create a GraphQL server.

We will use the preceding GraphQL concepts to create a GraphQL API server for the PhoneBook App using Oak and Oak-GraphQL modules in the upcoming chapter.

CHAPTER 8

PhoneBook App—API Implementation

Introduction

We will implement **Graph Query Language** (**GraphQL**) **Application Programming Interface** (**API**) server in this chapter. The basics of GraphQL we learnt previously will be applied in this chapter for API implementation. We will use **`oak`** and **`oak-graphql`** modules (introduced previously) to implement the API server.

Structure

In this chapter, we will cover the following topics:

- Initial setup
    - Version control using git
    - Trex as the package manager
    - Velociraptor as script runner
- Oak GraphQL Server setup
- Schema
- Resolvers
- DenoDB setup
    - DB
        - ElephantSQL PostgreSQL setup
    - Models
        - User
        - Contact
    - Controllers

- Authentication

Objective

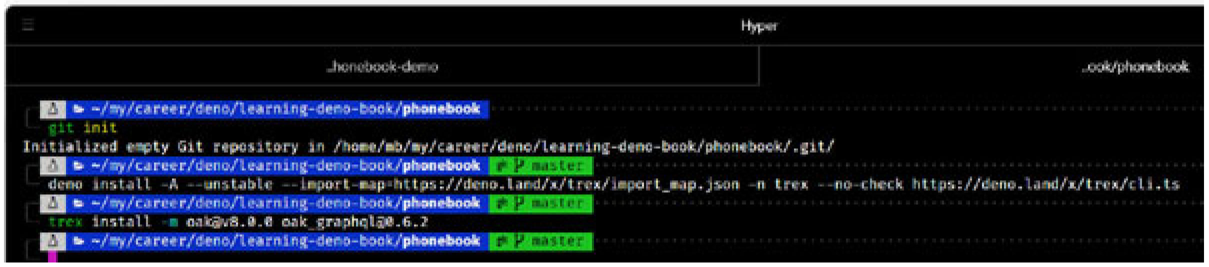
After completion of this chapter, you will be able to create GraphQL API server using **`Oak`** and **`Oak-GraphQL`** modules. You will learn the use of **Object-relational mapping** (**ORM**) using the **`denodb`** module to connect to different databases such as MySQL, PostgreSQL, NoSQL, and so on. You will also learn the use of **JSON Web Token** (**JWT**) to authenticate users using **`djwt`** module. You will also learn to setup a Deno server project using **`trex`** and **`velociraptor`** modules.

Initial setup

We will start coding by initiating a git repository. Then we shall install the **`oak`** and **`oak-graphql`** modules for creating Web server. We will use **`trex`** package management module for installing these modules.

Version control using git

Figure 8.1 shows git repo initialization, installation of **`trex`** module, and installation of **`oak`** and **`oak-graphql`** using **`trex`** module.

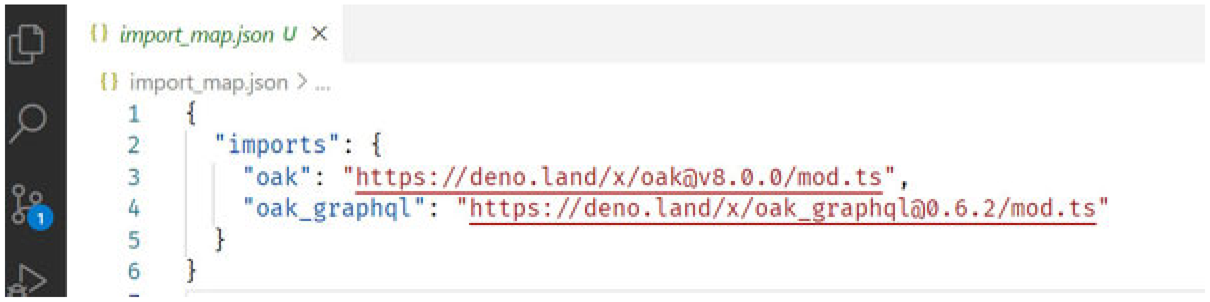


```
 ~/my/career/deno/learning-deno-book/phonebook
 git init
Initialized empty Git repository in /home/mb/my/career/deno/learning-deno-book/phonebook/.git/
 ~/my/career/deno/learning-deno-book/phonebook  master
 deno install -A --unstable --import-map=https://deno.land/x/trex/import_map.json -n trex --no-check https://deno.land/x/trex/cli.ts
 ~/my/career/deno/learning-deno-book/phonebook  master
 trex install -m oak@v8.0.0 oak_graphql@0.6.2
 ~/my/career/deno/learning-deno-book/phonebook  master
```

Figure 8.1: *Initial setup*

Trex as the package manager

Trex caches installed packages and add them to the **import_map.json** file. *Figure 8.2* shows the list of installed modules (**oak**, **oak-graphql**) in the **import_map.json** file.



```
{
  "imports": {
    "oak": "https://deno.land/x/oak@v8.0.0/mod.ts",
    "oak_graphql": "https://deno.land/x/oak_graphql@0.6.2/mod.ts"
  }
}
```

Figure 8.2: *Installed modules in import_map.json*

The installed modules can then be accessed as follows:

```
import { Application } from "oak";
```

Velociraptor as script runner

Create a file named **mod.ts** in **src** folder with the following code:

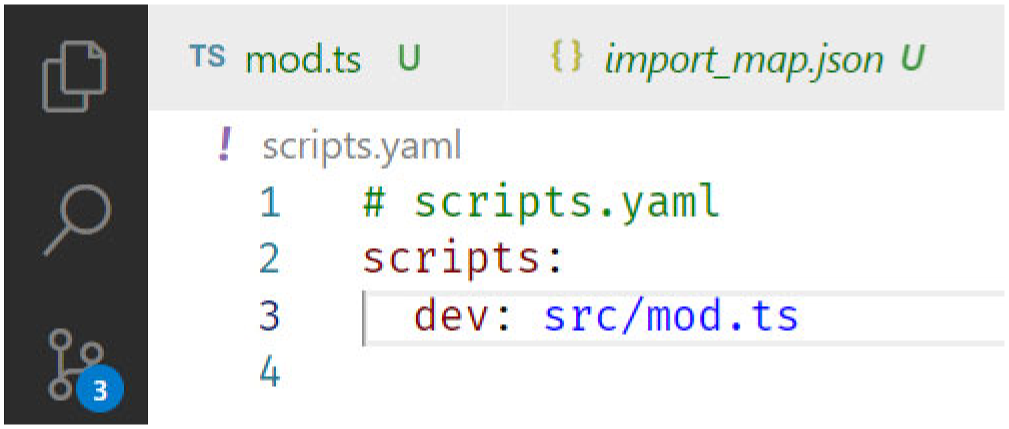
```
console.log("Hello Velociraptor!");
```

What is velociraptor, you would ask. It is a script runner module for Deno. We will install velociraptor and run the **mod.ts** file using it. Velociraptor can be installed using **https://nest.land/** package registry as follows:

```
deno install -qAn vr https://x.nest.land/velociraptor@1.1.0/cli.ts
```

We will now create **scripts.yaml** file to add the script to run the **mod.ts** file. Velociraptor detects files named **scripts** or **velociraptor** with extension **yml**, **yaml**, **json**, and **ts** as script files.

Figure 8.3 shows **scripts.yaml** file with **dev** script that executes **src/mod.ts** file.



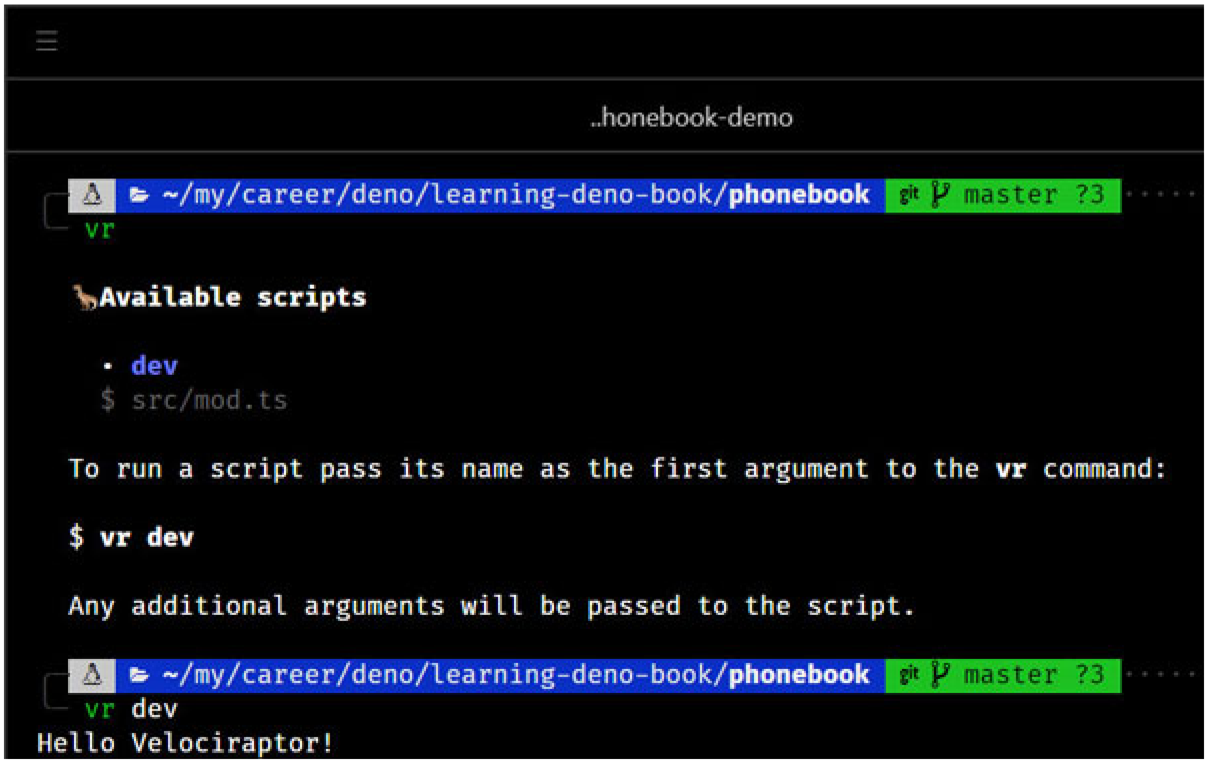
```yaml
# scripts.yaml
scripts:
  dev: src/mod.ts
```

Figure 8.3: *scripts.yaml file*

If we run **vr** command in the terminal, we will see a list of available scripts. We can then run the available script as follows:

1. `vr dev`

Figure 8.4 shows commands to list available scripts and run a script.



```
~/my/career/deno/learning-deno-book/phonebook  git master ?3
vr

Available scripts

  • dev
  $ src/mod.ts

To run a script pass its name as the first argument to the vr command:

$ vr dev

Any additional arguments will be passed to the script.

~/my/career/deno/learning-deno-book/phonebook  git master ?3
vr dev
Hello Velociraptor!
```

Figure 8.4: *Velociraptor CLI*

We have setup a Deno server project using trex and velociraptor. Now, we will set up a Web server in the next section.

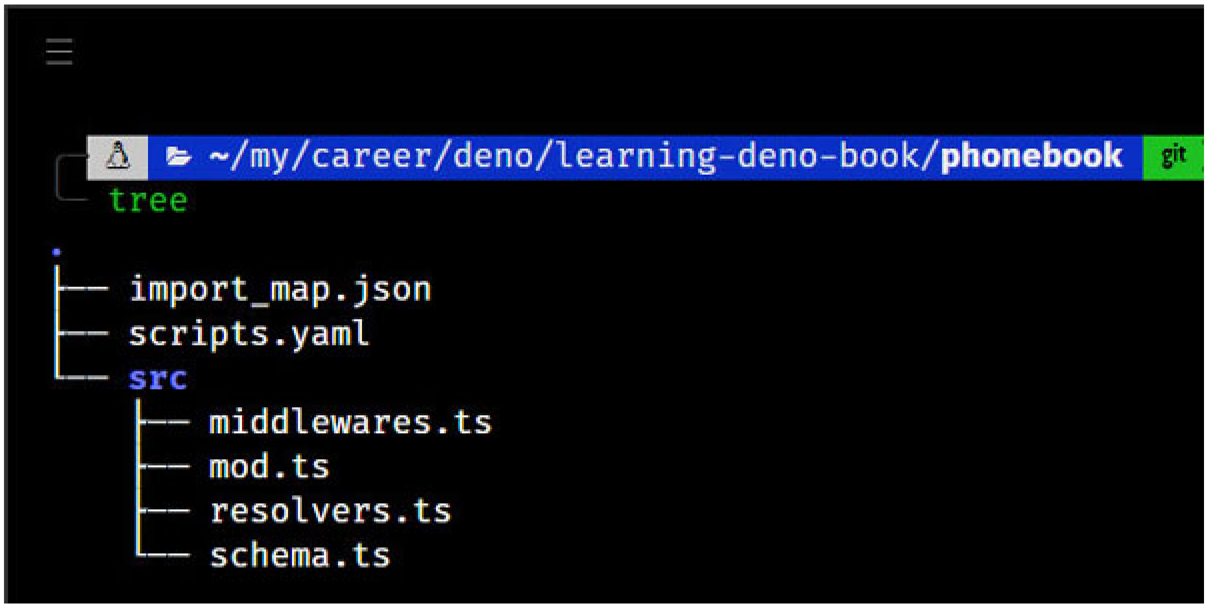
Oak GraphQL server setup

We will setup a simple GraphQL server using **`oak`** and **`oak-graphql`** modules. We will use **`oak-graphql`** code from previous *Chapter 7, Introduction to GraphQL, Oak, and Oak-GraphQL* to setup the server with few changes. We no longer need to mention the module URL. Instead, we can mention the module name mentioned in the **`import_map.json`** file.

```
1. import { Application, Router } from “oak”;
2. import { applyGraphQL, gql, GQLError } from “oak_graphql”;
```

We will structure the project by having **`mod.ts`** (main file), **`middlewares.ts`** (GraphQL Middleware), **`schema.ts`** (GraphQL schema/type definitions), and **`resolvers.ts`** (GraphQL Resolver) files.

Figure 8.5 shows the project structure with the preceding files:



```
 ~/my/career/deno/learning-deno-book/phonebook  git
tree
.
├── import_map.json
├── scripts.yaml
└── src
    ├── middlewares.ts
    ├── mod.ts
    ├── resolvers.ts
    └── schema.ts
```

Figure 8.5: *Project structure*

We will write code for each file one by one, starting from the **`mod.ts`** file as follows:

```
// src/mod.ts
import { Application} from “oak”;
import {GraphQLService} from “./middlewares.ts”

const app = new Application();

app.use(GraphQLService.routes(), GraphQLService.allowedMethods());

console.log(“Server listening on http://localhost:8080/graphql”);
await app.listen({ port: 8080 });
```

In the preceding code, we are creating oak server with GraphQL Middleware called **`GraphQLService`** that can be accessed at **`/graphql`** route.

```
// src/schema.ts
import { gql } from “oak_graphql”;

export const types = gql`
  type Query {
    hello(name: String): String
  }
`;
```

In the **`schema.ts`** file, we have defined a **`hello`** query that accepts **`name`** argument. GraphQL validates incoming requests according to data models defined in **`types`**. The incoming request will then be resolved by **`resolvers`**.

```
// src/resolvers.ts
import { GQLError } from “oak_graphql”;

export const resolvers = {
  Query: {
      hello: (parent: any, { name }: any, context: any, info: any) => {
      if (!name) throw new GQLError(“Name not provided”);
      return `Hello ${name}`;
    },
  },
};
```

We need to update **`scripts.yaml`** file for **`import_map`** file path and permissions. The file after the update will have the following code:

```
# scripts.yaml
imap: import_map.json
unstable: true
allow:
  - net
scripts:
  dev:
    cmd: src/mod.ts
    desc: “run graphql server in dev mode”
    watch: true
```

In the preceding code, **`imap`** points to **`import_map`** file path, **`unstable`** flag allows unstable Deno features to be executed, and **`allow`** includes **`net`** for allowing network access. We have also added **`cmd`** (execution command)**,** **`desc`** (description), and **`watch`** (file watcher) fields to **`dev`** command.

Schema

Schema is a type of system that defines the shape of data. It provides available data fields, queries, and mutations to clients. We have already learned schema-related concepts in the following chapter with code. We will add **types**, **queries**, and **mutations** for **`User`** and **`Contact`** entities as shown in the following code:

```
// src/schema.ts
import { gql } from “oak_graphql”;

export const types = gql`
  type User {
    userId: String!
    name: String!
    contacts: [Contact!]!
    email: String!
  }

  type Contact {
    contactId: String!
    name: String!
    phone: String
    email: String
    address: String
    user: User!
  }

```

```
type Login {
  token: String!
}

input RegisterUserInput {
  name: String!
  email: String!
  password: String!
}

input LoginUserInput {
  email: String!
  password: String!
}

input UpdateUserInput {
  name: String
}

input CreateContactInput {
  name: String!
  phone: String
  email: String
  address: String
}

input UpdateContactInput {
  contactId: String!
  name: String
```

```
    phone: String
    email: String
    address: String
  }

  type Query {
    user: User

    contacts: [Contact]
    contact(contactId: String!): Contact

    login(user: LoginUserInput): Login
  }

  type Mutation {
    registerUser(user: RegisterUserInput!): User
    deleteUser: Boolean
    updateUser(user: UpdateUserInput!): Boolean

    addContact(contact: CreateContactInput!): Contact
    deleteContact(contactId: String!): Boolean
    updateContact(contact: UpdateContactInput!): Boolean
  }
`;

```

We have defined various **types** and **inputs** in the preceding code. Let us explain them one by one.

User type

We are defining fields **`userId`**, **`name`**, **`contacts`**, and **`email`** for this type. The client receives user-related data as per this type. Let us explain each one of the fields. An exclamation mark “!” after the field type denotes a non-nullable type.

- **`userId`**: identifier or primary key
- **`name`**: user’s name
- **`contacts`**: user’s contacts in array
- **`email`**: user’s email

Contact type

We are defining fields **`userId`**, **`name`**, **`contacts`**, and **`email`** for this type. The client receives user-related data as per this type. Let us explain each one of the fields. An exclamation mark “!” after the field type denotes a non-nullable type.

- **`contactId`**: Identifier or primary key
- **`name`**: Name of the contact
- **`phone`**: Phone number of the contact
- **`email`**: Email of the contact
- **`address`**: Address of the contact
- **`user`**: userId of the creator of the contact

Login type

We are defining a **`token`** field in this type.

- **token:** Access token used to access protected API endpoints and identify requesting users.

RegisterUser Input

We are defining **`name`**, **`email`**, and **`password`** in this input type. It will be used as a **`mutation`** to create a user.

- **`name`**: Name of the user

- **`Email`**: Email of the user.
- **`Password`**: Password of the user.

LoginUser Input

We are defining **`email`** and **`password`** in this input type. Email and password created during registration will be used to authenticate the user and return the token.

- **`Email`**: Email of the user.
- **`Password`**: Password of the user.

UpdateUser Input

We are defining a single field **`name`** for this input. It will be used to update user information.

CreateContact and UpdateContact Input

Here, we are defining **`name`**, **`email`**, **`phone`**, and **`address`** fields for these types. Both types have identical fields, whereas **`UpdateContact`** has an extra field **`contactId,`** which is used to identify the contact to be updated.

Query Type

We have defined **`login`**, **`user`**, **`contacts`**, and **`contact`** queries for reading operations. **`Login`** query accepts input **user** of type **`LoginUserInput`** and returns data of **`Login`** type. **`user`** query returns details of the currently authenticated user.
`contacts` query returns an array of contacts added by the current user. It returns an array of **`Contact`** type. **`Contact`** query returns single **Contact** data using contactId.

Mutation Type

We have defined **`registerUser, deleteUser, updateUser, addContact, deleteContact`**, and **`updateContact`** queries for write operations. These queries perform create, update, and delete operations on User and Contact entities.

Resolvers

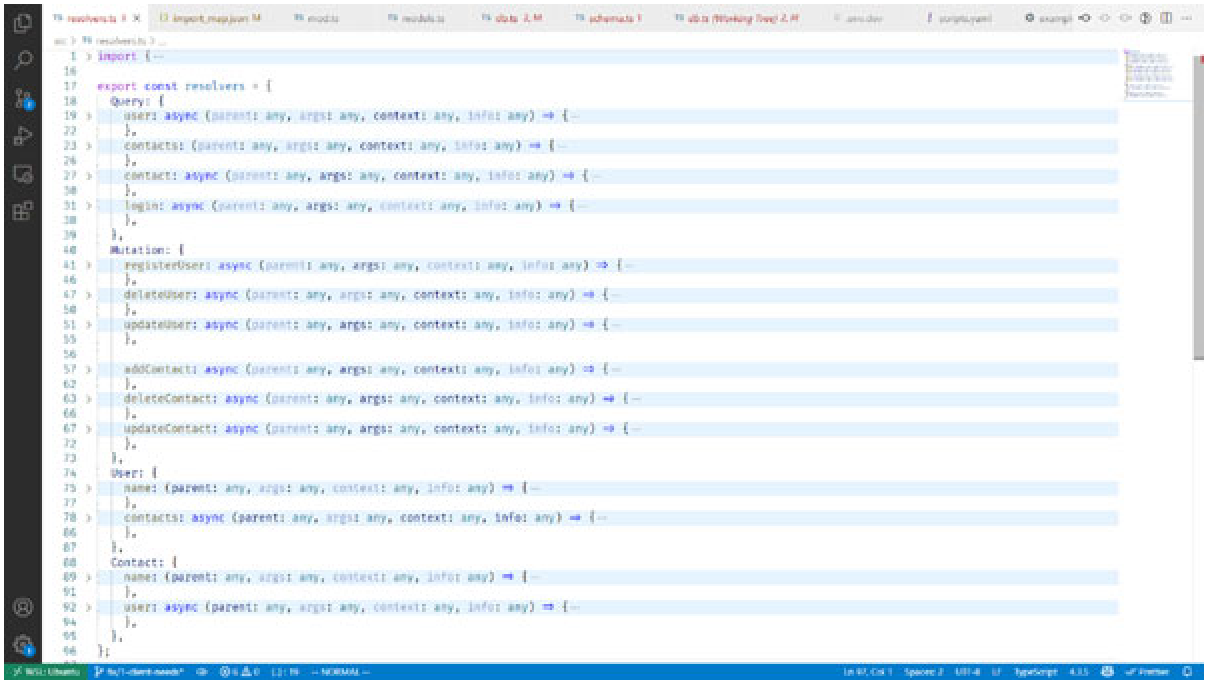
Queries and mutations defined in the schema previously will be *resolved* by the resolver for the respective query/mutation. Each resolver will perform database operations and return data for the query/mutation. The code for each resolver is as follows:

```
import {
  getUser,
  getUserByEmail,
  createUser,
  deleteUser,
  updateUser,
  getContactsByUserId,
  getContact,
  createContact,
  deleteContact,
  updateContact,
  userLoader,
} from “./controllers.ts”;
import { authenticate } from “./auth.ts”;
import { GQLError } from “oak_graphql”;

export const resolvers = {
  Query: {
      user: async (parent: any, args: any, context: any, info: any) => {
```

```
      if (!context?.userId) throw new GQLError(“Authentication error”);
      return getUser(context.userId);
    },
    contacts: (parent: any, args: any, context: any, info: any) => {
      if (!context?.userId) throw new GQLError(“Authentication error”);
      return getContactsByUserId(context.userId);
    },
    contact: async (parent: any, args: any, context: any, info: any) => {
      if (!context?.userId) throw new GQLError(“Authentication error”);
      return getContact(args.contactId);
    },
    login: async (parent: any, args: any, context: any, info: any) => {
      const { email, password } = args.user;
    const user = await getUserByEmail(email);
      if (!user) throw new GQLError(`Invalid email or password`);
      const authToken = await authenticate(user, password);
      if (!authToken) throw new GQLError(`Invalid email or password`);
      return { token: authToken };
    },
  },
  Mutation: {
    registerUser: async (parent: any, args: any, context: any, info: any) => {
      const { name, email, password } = args.user;
      const user = await createUser(name, email, password);
      console.log(“created user :>> “, user);
      return user;
    },
    deleteUser: async (parent: any, args: any, context: any, info: any) => {
      if (!context?.userId) throw new GQLError(“Authentication error”);
      return deleteUser(context.userId);
    },
    updateUser: async (parent: any, args: any, context: any, info: any) => {
      if (!context?.userId) throw new GQLError(“Authentication error”);
      const { name, isActive } = args.user;
      return updateUser(context.userId, { name, isActive });
    },

    addContact: async (parent: any, args: any, context: any, info: any) => {
      if (!context?.userId) throw new GQLError(“Authentication error”);
      const contact = await createContact(args.contact, context.userId);
      console.log(“created user :>> “, contact);
      return contact;
    },
    deleteContact: async (parent: any, args: any, context: any, info: any) => {
      if (!context?.userId) throw new GQLError(“Authentication error”);
      return deleteContact(args.contactId);
    },
    updateContact: async (parent: any, args: any, context: any, info: any) => {
      if (!context?.userId) throw new GQLError(“Authentication error”);
      console.log(`args`, args);
      const { contactId, …contactData } = args.contact;
      if (contactId) return updateContact(contactId, contactData);
    },
  },
  User: {
    name: (parent: any, args: any, context: any, info: any) => {
      return parent.name;
    },
    contacts: async (parent: any, args: any, context: any, info: any) => {
      const contacts = await resolvers.Query.contacts(
        parent,
        {},
        context,
        info
      );
      return contacts;
    },
  },
  Contact: {
    name: (parent: any, args: any, context: any, info: any) => {
      return parent.name;
    },
    user: async (parent: any, args: any, context: any, info: any) => {
      return await userLoader.load(parent.userId);
    },
  },
};

```

Our code for resolvers is based on the resolver code from the previous chapter. We have defined **`Query`**, **`Mutation`**, **`User`**, and **`Contact`** objects, each containing resolvers. **`Query`** object contains **`user,`** **`contacts`**, **`contact`**, and **`login`** resolvers**.** **`Mutation`** contains **`registerUser`**, **`updateUser`**, **`deleteUser`**, **`addContact`**, **`updateContact`**, and **`deleteContact`** resolvers. The other objects, **`User`** and **`Contact`**, are useful for resolving internal fields (name, and so on) of the respective type. The data can then be fetched from various sources. Each resolver function accepts optional parameters **`parent,`** **`args`**, **`context`**, and **`info`**, which are already explained in the previous chapter.

Figure 8.6 shows a list of all the resolvers from the **`resolvers.ts`** file



```
import {…
 
export const resolvers = {
  Query: {
    user: async (parent: any, args: any, context: any, info: any) => {…
    },
    contacts: (parent: any, args: any, context: any, info: any) => {…
    },
    contact: async (parent: any, args: any, context: any, info: any) => {…
    },
    login: async (parent: any, args: any, context: any, info: any) => {…
    },
  },
  Mutation: {
    registerUser: async (parent: any, args: any, context: any, info: any) => {…
    },
    deleteUser: async (parent: any, args: any, context: any, info: any) => {…
    },
    updateUser: async (parent: any, args: any, context: any, info: any) => {…
    },

    addContact: async (parent: any, args: any, context: any, info: any) => {…
    },
    deleteContact: async (parent: any, args: any, context: any, info: any) => {…
    },
    updateContact: async (parent: any, args: any, context: any, info: any) => {…
    },
  },
  User: {
    name: (parent: any, args: any, context: any, info: any) => {…
    },
    contacts: async (parent: any, args: any, context: any, info: any) => {…
    },
  },
  Contact: {
    name: (parent: any, args: any, context: any, info: any) => {…
    },
    user: async (parent: any, args: any, context: any, info: any) => {…
    },
  },
};
```

Figure 8.6: *Resolvers*

While resolving a query, we need to perform operations such as read/write database, authentication, and so on. We are handling database operations in the **`controllers.ts`** file and authentication in the **`auth.ts`** file.

We will perform, create, update, and delete operations on database models **`User`** and **`Contact`** defined in the **`models.ts`** file. We need to set up a database and link it with the models in order to use the models in controllers. Let us set up a database and create models using the Deno module named **`denodb`**.

Database setup

We will set up a PostgreSQL database using **`ElephantSQL`**. Head over to https://www.elephantsql.com/ and login/signup with your account. Once done, we need to create an instance named **`phonebook`** with a **tiny turtle** (free) plan. Open the created instance and copy the **`URL`** mentioned in the **`Details`** section. We will set an environment variable named **`DATABASE_URL`** in the **`.env`** file as follows:

```
//  .env
// Replace following url with elephantsql url
DATABASE_URL=postgres://deno_admin:deno_admin@localhost:5432/deno_db
```

Now, to access **`DATABASE_URL`** in the code, we will add **`envFile`** flag with **`.env`** file in **`scripts.yaml`** file as follows:

```
# scripts.yaml
…
allow:
  - net
  - env
envFile:
  - .env
…
```

Install **`denodb`** using **`trex i -c https://deno.land/x/denodb/mod.ts`** command and use **`PostgresConnector`** to connect with the database URL in the **`db.ts`** file as follows:

```
import { Database, PostgresConnector } from “denodb”;

const DATABASE_URL = Deno.env.get(“DATABASE_URL”) as string;

const connector = new PostgresConnector({
  uri: DATABASE_URL,
});

export const db = new Database(connector);
```

The preceding database will be used and linked with database models **`User`** and **`Contact`** in the next section.

Models

We will define two models, **`User`** and **`Contact`**, using **Model, DataTypes,** and **Relationships** from **`denodb`** module. The code for both models is as follows:

```
// src/models.ts

import { Model, DataTypes, Relationships } from “denodb”;
import { db } from “./db.ts”;
import { v4 } from “uuid”;

export class User extends Model {
  static table = “users”;
  static timestamps = true;

  static fields = {
    userId: {
      type: DataTypes.STRING,
      primaryKey: true,
    },
    name: { type: DataTypes.STRING, length: 50 },
    isActive: { type: DataTypes.BOOLEAN },
    email: { type: DataTypes.STRING },
    password: { type: DataTypes.STRING },
  };

  static defaults = {
    userId: v4.generate,
    name: “ABC”,
    isActive: true,
  };

  static contacts() {
    return this.hasMany(Contact);
  }
}

export class Contact extends Model {
  static table = “contacts”;
  static timestamps = true;

  static fields = {
    contactId: {
      type: DataTypes.STRING,
      primaryKey: true,
    },
    name: { type: DataTypes.STRING, length: 100 },
    phone: { type: DataTypes.STRING, length: 50, allowNull: true },
    email: { type: DataTypes.STRING, length: 100, allowNull: true },
    address: { type: DataTypes.STRING, length: 200, allowNull: true },
  };

  static defaults = {
    contactId: v4.generate,
    name: “Learn Deno”,
    isCompleted: false,
  };

  static user() {
    return this.hasOne(User);
  }
}

Relationships.belongsTo(Contact, User);

db.link([User, Contact]);
await db.sync({ drop: true });
```

For both models, we have defined **`table`**, **`timestamps`**, **`fields`**, **`defaults`,** and **`relationship`** static properties**.** The **`table`** will be the name of the database table for the model, e.g., **`users`** from **`User`** model and **`contacts`** for **`Contact`** model. We will set **`timestamps`** to true in order to add **`created_at`** **and** **`updated_at`** fields to the model.

The **`fields`** property will contain all the model fields defined with the **`type (INTEGER, STRING,`** and so on**)**, **`length (max length)`**, **`allowNull (nullable)`**, and **`primaryKey`**. We can also provide default values to the fields using the **`defaults`** property. Here, we are using the **`v4.generate`** function provided by **uuid** (standard Deno module) to generate **`UUID`** for **`userId`** **and** **`contactId`** fields. Make sure you have installed **uuid** module using the following command:

```
trex -c uuid=https://deno.land/std@0.132.0/uuid/mod.ts
```

Then, we added query methods to query relationship value **`contacts`** for User and **`user`** for contact**.**

- **`this.hasMany(Contact)`** will return **`Contact`** instances where **`Contact.userId`** matches **`User.userId`**.

- **`this.hasOne(User)`** will return **`User`** instance where **`User.userId`** matches **`Contact.userId`**.

We will then add a foreign key using the **`Relationships.belongsTo`** method. The database link is done afterward using the **`link`** method. Finally, models are synchronized using the **`sync`** method with a **`drop`** flag to remove existing records.

Controllers

We need to map resolvers with models via controllers. Each controller will perform database operations on models to get, create, update, and delete an entity. All the controller functions are added in the **`controllers.ts`** file as follows:

```
import { User, Contact } from “./models.ts”;
import { GQLError } from “oak_graphql”;
import DataLoader from “dataloader”;
import { hash } from “scrypt”;

interface IContact {
  name: string;
  phone: string;
  email: string;
  address: string;
}

export const getUser = async (userId: string) => {
  return await User.find(userId);
};

export const getUserByEmail = async (email: string) => {
  const existingUser = await User.where(“email”, email).first();
  return existingUser;
};

export const createUser = async (
  name: string,
  email: string,
  password: string
) => {
  const existingUser = await getUserByEmail(email);
  if (existingUser) throw new GQLError(“User already exists”);
  const hashedPassword = await hash(password);
  const user = await User.create({ name, email, password: hashedPassword });
  return user;
};

export const deleteUser = async (userId: string) => {
  await Contact.where(“userId”, userId).delete();
  await User.deleteById(userId);
  return null;
};

export const updateUser = async (userId: string, userData: any) => {
  await User.where(“userId”, userId).update(userData);
  return null;
};
```

```

export const getContactsByUserId = async (userId: string) => {
  return await User.where(“userId”, userId).contacts();
};

export const getContact = async (contactId: string) => {
  return await Contact.find(contactId);
};

export const createContact = async (contactData: IContact, userId: string) => {
  const existingUser = await User.where(“userId”, userId).get();
  if (!existingUser.length) throw new GQLError(“User does not exist”);
  const contact = await Contact.create({ …contactData, userId });
  return contact;
};

export const deleteContact = async (userId: string) => {
  await Contact.deleteById(userId);
  return null;
};

export const updateContact = async (contactId: string, contactData: any) => {
  await Contact.where(“contactId”, contactId).update(contactData);
  return null;
};

const batchGetUsersByIds = (ids: any) => {
  const userPromises = ids.map(getUser);
  const usersPromise = Promise.resolve(userPromises);
  return usersPromise;
};

export const userLoader = new DataLoader(batchGetUsersByIds);
```

In the preceding code, we have defined **getUser**, **getUserByEmail**, **createUser**, **deleteUser,** and **updateUser** functions for **User** model related queries and **getContactByUserId**, **getContact**, **updateContact**, and **deleteContact** functions for Contact model related queries. We are using **find**, **where**, **create**, **deleteById**, **update**, and **delete** model methods to perform the operations.

We have also defined **batchGetUsersByIds** function that works as **dataloader** function. It collects all the user's ids and all the users in a single database call. It then returns a promise that resolves the array. We need to install **dataloader** using the following command to enable this functionality.

```
trex -c dataloader=https://esm.sh/dataloader@2.0.0/index.js
```

Also, we are hashing the password (transforming the password into another string) in **createUser** function using the **hash** function provided by **scrypt** module. We will install scrypt module using the following command:

```
trex i -c https://deno.land/x/denOBodb/mod.ts
```

Authentication

In order to identify requesting users, we will use **`djwt`** module to create and verify **JSON Web Token (JWT).** We need to set a **`SECRET_KEY (random string)`** in **`.env`** file, which will be used during JWT creation and verification. All the authentication-related code is written in the **`auth.ts`** file as follows:

```
// src/auth.ts

import { create, verify } from “djwt”;
import { RouterContext } from “oak”;
import { verify as hashVerify } from “scrypt”;

const SECRET_KEY = Deno.env.get(“SECRET_KEY”);
const ALGORITHM = “HS512”;


export const authenticate = async (user: any, password: string) => {
  if (await verifyPassword(password, user.password)) {
    return createToken(user);
  }
};

const verifyPassword = async (password: string, hashedPassword: string) => {
  const isValidPassword = await hashVerify(password, hashedPassword as string);
  return isValidPassword;
};

```

```
const createToken = async (user: any) => {
  if (!SECRET_KEY)
    throw new Error(“SECRET_KEY not set as environment variable”);
  const jwt = await create(
    { alg: ALGORITHM, typ: “JWT” },
    { userId: user.userId },
    SECRET_KEY
  );
  return jwt;
};

export const setContextUser = async (ctx: RouterContext) => {
  const token = await parseToken(ctx);
  if (!token) return null;
  const user = await verifyToken(token);
  return user;
};
const parseToken = async (ctx: RouterContext) => {
  const authorizationHeader = ctx.request.headers.get(“authorization”);
  const parts = authorizationHeader ? authorizationHeader.split(“ “) : [];
  const token =
    parts.length === 2 && parts[0].toLowerCase() === “bearer”
      ? parts[1]
      : undefined;
  return token;
};

const verifyToken = async (token: string | undefined) => {
  if (token) {
    if (!SECRET_KEY)
      throw new Error("SECRET_KEY not set as environment variable");
      const payload = await verify(token, SECRET_KEY, ALGORITHM);
    return payload;
  }
  return;
};
```

In the **`authenticate`** function, we are accepting **`user`** and **`password`** as arguments. We are using **`verifyPassword`** function to verify the password by checking it against the hashed password. If the password is valid, we then create a *JWT token* using **`SECRET_KE`** and **`ALGORITHM`** with payload **`userId`**.

In the **`setContextUser`** function, we are accepting **`ctx`** of type **`RouterContext`** as an argument. We then pass the ctx to **`parseToken`** function that returns a **`token`** provided in the **authorization** header. The parsed token is then passed to **`verifyToken`** function that verifies the token using **`SECRET_KEY`** and **`ALGORITHM`** and returns the **payload** after successful verification. The payload will then be available for all the resolvers to identify the requesting user. We need to install **`djwt`** module using the following command:

```
trex i -m djwt@v2.2
```

We will now use **`setContextUser`** as a context handler for GraphQL middleware in the **`middlewares.ts`** file as follows:

```
// src/middlewares.ts
```

```

import { Router } from “oak”;
import { applyGraphQL } from “oak_graphql”;
import { types } from “./schema.ts”;
import { resolvers } from “./resolvers.ts”;
import { setContextUser } from “./auth.ts”;

export const GraphQLService = await applyGraphQL<Router>({
  Router,
  typeDefs: types,
  resolvers: resolvers,
  context: setContextUser,
});
```

In the **`src/middlewares.ts`** file, we are importing GraphQL schema/type definitions from the **`types.ts`** file. We are also importing resolvers from the **`resolvers.ts`** file. They are passed along with **Router** to **`applyGraphQL`** service to create **`GraphQL`** service. The **`setContextUser`** function uses **JSON Web Token (JWT)** provided in the authorization header to retrieve userId and set it in the context so every resolver will have access to it.

Finally, we will add **`oakcors`** middleware in our app to enable **cross-origin resource sharing (CORS)** to allow other domains to access the resources on this server. We will also use the **PORT** environment variable to set the server port. Our **`mod.ts`** file will have the following code after changes:

```
import { Application } from “oak”;
import { GraphQLService } from “./middlewares.ts”;
import { oakCors } from “cors”;

```

```
const app = new Application();

const _PORT = Deno.env.get(«PORT»);
const PORT = parseInt(_PORT as string);
const port = isNaN(PORT) ? 8080 : PORT;

app.use(
  oakCors()
);

app.use(GraphQLService.routes(), GraphQLService.allowedMethods());

console.log(`GraphQL Server listening on ${port}/graphql`);

await app.listen({ port });
```

We will install **`cors`** module using the following command:

```
trex i -m cors@v1.2.2
```

Conclusion

We have implemented a GraphQL API in this chapter using various Deno modules such as **`oak-graphql`**, **`djwt`**, **`scrypt`**, and **`cors`**. Oak-GraphQL is useful for the creation of a GraphQL API server along with the Oak framework. Djwt is useful to create and verify JSON Web Tokens. Scrypt is useful for hashing passwords. CORS is useful for allowing server access via different domains. We learnt how to configure settings (env variables, scripts, and permissions) using *velociraptor*. We used **`trex`** module to install Deno modules and add them to **`import_map.json`** file.

This knowledge will be helpful in setting up a new Deno project. We can test the implementation in the playground for now. We will create a Svelte front end using **`Snel`** module and integrate the GraphQL API in the upcoming chapter.

CHAPTER 9

PhoneBook App - UI Implementation

Introduction

We implemented the GraphQL API server in the previous chapter for the PhoneBook App. The **User Interface** (**UI**) for the application will be implemented in this chapter using that API. We will be using `snel` module to create `svelte` applications in deno along with `snel-carbon` module which provides carbon components and icons based on carbon design system by IBM. We will also use axiod module to make network requests to the server.

Structure

In this chapter, we will cover the following topics:

- Initial setup
  - Snel setup
  - Snel-carbon installation
- Routes
  - Public routes
    - Login
    - Register
  - Private routes
    - Home
    - Contacts
      - ContactTable
      - ContactForm

- Layout
    - Header
    - Sidebar

Objective

After completion of this chapter, you should be able to create svelte UI applications using **`snel`** and **`snel-carbon`** modules. You should be able to create public and private routes for svelte applications afterwards. You will also learn the use of **`axiod`** module to make network requests.

Initial setup

In this section, we will install **`snel`** and initialize a snel project and then use **`trex`** to run the scripts. Once a snel project is created, we will install **`snel-carbon`** components, icons module and **`axiod`** module for network requests. Let's start with snel installation.

Snel setup

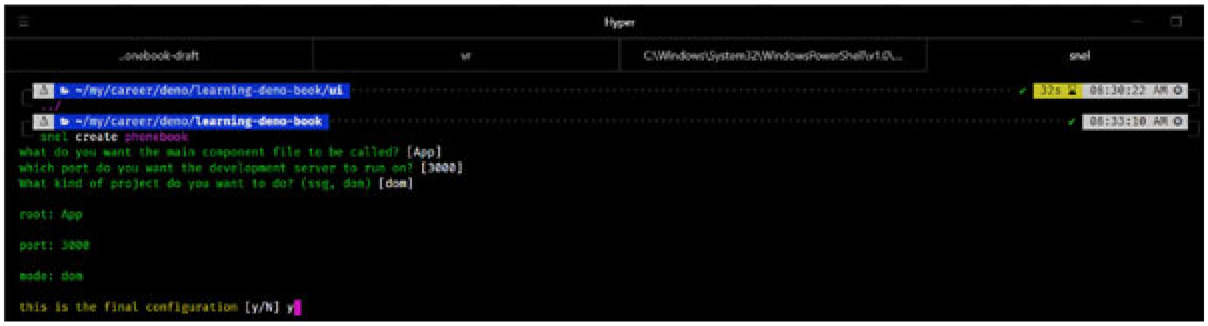
Snel is "*A Cybernetical framework for svelte applications in deno*" that compiles Svelte components to javascript files. We will install snel using following command:

```
1. deno run --allow-run --allow-read https://deno.land/x/snel/install.ts
```

Above command will install **`snel`**, **`trex`**, **`bundler`** modules to provide good developer experience. Once the command is successfully executed, we will create new snel app using following command:

```
1. snel create [project name]
```

Figure 9.1 shows creation of new snel project with default settings.

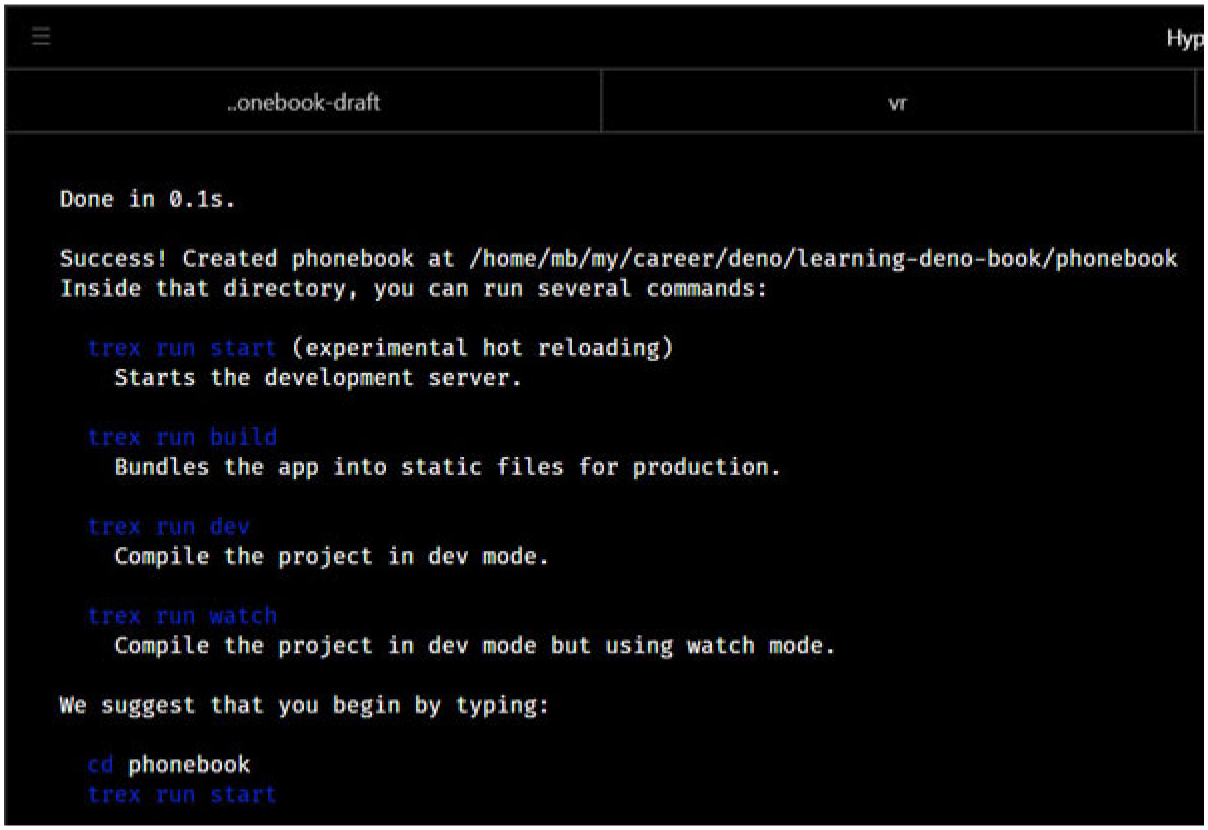


```
~/my/career/deno/learning-deno-book/ui                        ✓ 32s  08:30:22 AM
../
~/my/career/deno/learning-deno-book                            ✓ 08:33:10 AM
  snel create phonebook
what do you want the main component file to be called? [App]
which port do you want the development server to run on? [3000]
What kind of project do you want to do? (ssg, dom) [dom]

root: App

port: 3000

mode: dom

this is the final configuration [y/N] y
```

Figure 9.1: *Snel Project Creation*

Once the project is created, we can use several commands to run the project in various modes like **`development, watch, build`**, and so on.

[*Figure 9.2*](#) lists several commands available to run the snel project.



```
Done in 0.1s.

Success! Created phonebook at /home/mb/my/career/deno/learning-deno-book/phonebook
Inside that directory, you can run several commands:

  trex run start (experimental hot reloading)
    Starts the development server.

  trex run build
    Bundles the app into static files for production.

  trex run dev
    Compile the project in dev mode.

  trex run watch
    Compile the project in dev mode but using watch mode.

We suggest that you begin by typing:

  cd phonebook
  trex run start
```

Figure 9.2: *Trex Commands for Snel Project*

We will start the project by running **`trex run start`** command as suggested in the message. This will run the server on port **`3000`**.

Figure 9.3 shows a sample snel project running on port 3000.

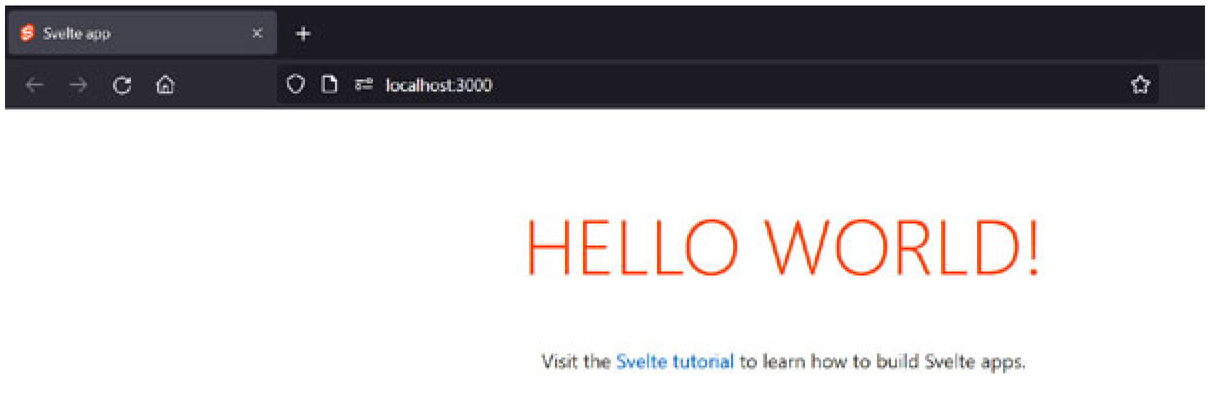

Svelte app
localhost:3000
HELLO WORLD!
Visit the Svelte tutorial to learn how to build Svelte apps.

Figure 9.3: *Snel sample project running on port 3000*

Snel Carbon installation

Snel-Carbon is a deno module providing UI components for svelte. It would help us to build a UI based on the Carbon Design System. Let's start by adding css in the **`index.html`** file. This will add stylings required for the components.

```
<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/carbon-components-svelte@0.44.2/css/all.min.css" />
```

Snel provides **`snel.config.js`** file to configure snel settings. It provides **`extendsImportMap`** field to link external import map to the project. We will add import maps for components and icons provided by **`snel-carbon`** library in this field as follows:

```
// snel.config.js

export default {
  port: 3000,
  mode: "dom",
  plugins: [],
  extendsImportMap: [
    “https://denopkg.com/crewdevio/Snel-carbon@main/components.map.json”,
    “https://denopkg.com/crewdevio/Snel-carbon@main/icons.map.json”,
  ],
};
```

We will now install **axiod** module using familiar **trex** command as follows:

```
trex i -m axiod@0.20.0-0
```

Above command will add **axiod** in **import_map.json** file in the project. We have installed all the required modules in this section. We will now create routes for different pages in the next section.

Routes

Snel module includes **Svelte Routing** library to create routes. We will use it to create two types of routes named public and private routes for our project. Public routes will be available to all the users while private routes will be available only to authenticated users.

Let’s start by adding **Route** component from **routes/index.svelte** file in **App.svelte** file as follows:

```
// App.svelte

<script>
  import Routes from “routes/index.svelte”;
  import { fade } from “svelte/transition”;
</script>
```

```

<main transition:fade>
  <Routes />
</main>

<style>
  main {
    max-width: 240px;
  }

  @media (min-width: 640px) {
    main {
      max-width: none;
    }
  }
</style>
```

In the preceding code, we are replacing code within the main element (HTML element that represents the dominant content of the body) by **`Route`** component. This way our app will always render each page as per the route address. Let's implement the routes in the **`routes/index.svelte`** file as follows:

```
<script>
  // Modules
  import Router from "svelte-routing/Router.svelte";
  import Route from "svelte-routing/Route.svelte";

  // Routes
  import Contacts from "./Contacts.svelte";
  import Login from “./Login.svelte”;
  import Register from “./Register.svelte”;
  import Home from “./Home.svelte”;
  import NotFound from “./NotFound.svelte”;
  import PrivateRoute from “./PrivateRoute.svelte”;

  import Content from “snel-carbon/components/UIShell/Content.svelte”;
  import Grid from “snel-carbon/components/Grid/Grid.svelte”;
  import Row from “snel-carbon/components/Grid/Row.svelte”;
  import Column from “snel-carbon/components/Grid/Column.svelte”;

  // Components
  import Layout from “@/components/Layout.svelte”;
</script>

<Router>
  <Layout />
  <Content>
    <Grid>
      <Row>
        <Column>
          <Route path=”login” component={Login} />
          <Route path=”register” component={Register} />
          <Route path=”*” component={NotFound} />

          <PrivateRoute path=”/”>
          <Home />
        </PrivateRoute>
        <PrivateRoute path="contacts" let:location>
          <Contacts />
        </PrivateRoute>

      </Column>
    </Row>
  </Grid>
</Content>
</Router>
```

In the preceding code, we are using **`Router`**, **`Route`** components from **`svelte-routing`** module. **`Router`** provides route information via context to **`Route`**, **`Link`** components and renders best matched **`Route`**.

In the **`Routes`** import section, we are importing **`Contacts`**, **`Login`**, **`Home`**, and **`Register`**. Each route will be assigned a path. Apart from these routes, we are also importing two special routes named **`NotFound`**, **`PrivateRoute`**. **`NotFound`** renders when no other route is matched. **`PrivateRoute`** is used to restrict access to routes like **`Contacts`** for authenticated users only.

We are importing Carbon components such as **`Content`**, **`Row`**, **`Grid`, `Column`** from **`snel-carbon`** module to create a responsive UI along with **Layout** custom component having header and sidebar implementation.

Let's implement each route code one by one.

Login

Login will be the default page shown to every unauthenticated user. We will use **`Form`** to handle validations for **`email`**, **`password`**. The implementation for **`Login.svelte`** is as follows:

```
// Login.svelte

<script>
  import { navigate } from “svelte-routing”;

  import Form from “snel-carbon/components/Form”;
  import Tile from “snel-carbon/components/Tile”;
  import TextInput from “snel-carbon/components/TextInput/TextInput.svelte”;
  import PasswordInput from “snel-carbon/components/TextInput/PasswordInput.svelte”;
  import Button from “snel-carbon/components/Button”;
  import InlineLoading from “snel-carbon/components/InlineLoading”;
  import InlineNotification from “snel-carbon/components/Notification/InlineNotification.svelte”;

  import axiod from “https://deno.land/x/axiod/mod.ts”;

  import { accessToken } from “../utils/stores.ts”;

  let email = “”;
  let password = “”;
  let loading = false;
  let errors = [];

  const handleSubmit = (e) => {
    loading = true;
    const query = `
      {
        login(user:{email:”${email}”, password:”${password}”}){
          token
        }
      }
`;
    axiod
      .post(“http://localhost:8080/graphql”, { query })
      .then((res) => {
        if (res.data.errors) {
          errors = res.data.errors;
        }
        if (res.data.data?.login?.token) {
          localStorage.setItem(“accessToken”, res.data.data?.login?.token);
          $accessToken = res.data.data?.login?.token;
          navigate(“/contacts”, {
            replace: false,
          });
        }
      })
      .catch((err) => {
        console.log(`err`, err);
      })
      .finally(() => {
        loading = false;
      });
```

```
  };
</script>

{#each errors as error (error.message)}
   <InlineNotification timeout={5000} type=”error” title={error.message} />
{/each}

<Tile>
  <Form on:submit={handleSubmit}>
    <TextInput
      bind:value={email}
      required
      type=”email”
      labelText=”Email”
      placeholder=”Enter Email”
      size=”sm”
    />
    <PasswordInput
      bind:value={password}
      required
      type=”password”
      labelText=”Password”
      placeholder=”Enter Password”
      size=”sm”
    />
    {#if loading}
      <InlineLoading />
    {:else}
```

```
      <Button type=”submit” size=”small”>Login</Button>
    {/if}
  </Form>
</Tile>
```

In the above code, we are using various **snel-carbon** components like **Form**, **Tile**, **TextInput**, **PasswordInput**, **Button**, **InlineLoading**, **InlineNotifications** to create the UI. We are using **navigate** function to redirect to **contacts** route on successful login. We are also importing **accessToken** which is stored in **stores.ts** file to check if the user is authenticated or not and **axiod** will be used to make network requests to GraphQL API Server.

We are using four states, **email**, **password**, **errors**, **loading** for this component where email, password will be used to store user input via **TextInput**, **PasswordInput**, loading will be used to render loader during network request using **InlineLoading** and errors will be used to store errors from the response.

In the UI part, the **TextInput** and **PasswordInput** are bound to **email**, **password** states respectively. The **Form** component will invoke **handleSubmit** function on **submit** event. The **handleSubmit** function makes network request using **axiod** with **login** query. If the response contains errors then those errors will be stored as **errors** and will be rendered using **InlineNotification** component. If the response provides **token** in the **data,** it will be stored into localStorage as well as **stores** as **accessToken** and the user will be redirected to **contacts** route.

Figure 9.4 shows login route with email, password input fields.

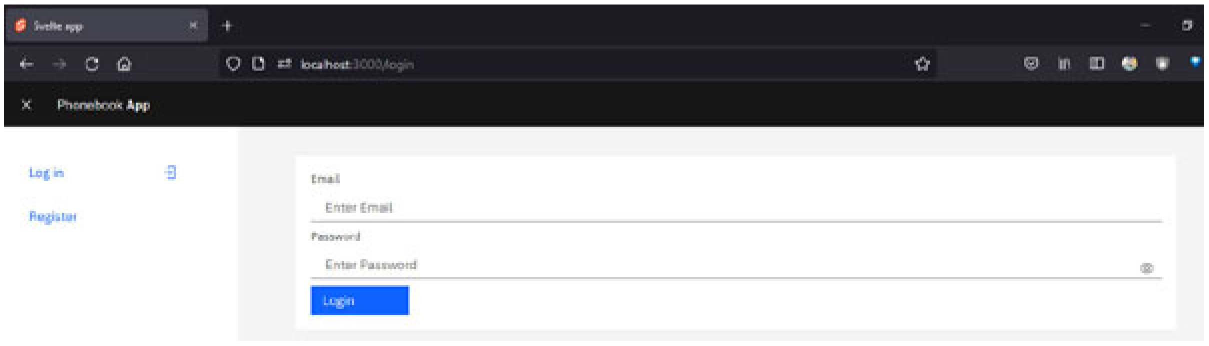


Figure 9.4: *Login Route*

Register

This route is similar to `login` route with few changes. It will have additional states and few changes in the query and the network response handling. The implementation for `Register.svelte` route is as follows:

```
<script>
  import { navigate } from “svelte-routing”;

  import Form from “snel-carbon/components/Form”;
  import Tile from “snel-carbon/components/Tile”;
  import TextInput from “snel-carbon/components/TextInput/TextInput.svelte”;
  import PasswordInput from “snel-carbon/components/TextInput/PasswordInput.svelte”;
  import Button from “snel-carbon/components/Button”;
  import InlineNotification from “snel-carbon/components/Notification/InlineNotification.svelte”;
  import InlineLoading from “snel-carbon/components/InlineLoading”;

  import axiod from “https://deno.land/x/axiod/mod.ts”;

  let name = “”;
  let email = “”;
  let password = “”;
  let confirmPassword = “”;
  let loading = false;
  let errors = [];

  const handleSubmit = (e) => {
    if (password === confirmPassword) {
      loading = true;
      const query = `
      mutation{
        registerUser(user:{email:”${email}”, password:”${password}”, name:”${name}”}){
          name
        }
      }
`;
      axiod
        .post(“http://localhost:8080/graphql”, { query })
        .then((res) => {
          console.log(`res`, res);
          if (res.data.errors) {
            errors = res.data.errors;
          }
          if (res.data.data.registerUser?.name) {
            alert(`user ${res.data.data.registerUser.name} registered`);
            navigate(“/contacts”, {
              replace: false,
              state: { accessToken: res.data.data.login },
            });
          }
        })
        .catch((err) => {
          console.log(`err`, err);
        });
    } else {
      alert(“Passwords do not match”);
    }
  };
</script>

{#each errors as error}
  <InlineNotification type=”error” title={error.message} />
{/each}

<Tile>
  <Form on:submit={handleSubmit}>
    <TextInput
      bind:value={name}
      required
      labelText=”Name”
      placeholder=”Enter Name”
      size=”sm”
    />
    <TextInput
```

```
      bind:value={email}
      required
      type=”email”
      labelText=”Email”
      placeholder=”Enter Email”
      size=”sm”
    />
    <PasswordInput
      bind:value={password}
      required
      type=”password”
      labelText=”Password”
      placeholder=”Enter Password”
      size=”sm”
    />
    <PasswordInput
      bind:value={confirmPassword}
      required
      type=”password”
      labelText=”Confirm Password”
      placeholder=”Enter Confirm Password”
      size=”sm”
    />
    {#if loading}
      <InlineLoading />
    {:else}
      <Button type=”submit” size=”small”>Register</Button>
    {/if}
  </Form>
```

```
</Tile>
```

As we can see, the above code has many similarities with **`Login.svelte`** code. We have added two additional states **`name`** and **`confirmPassword`** each bound to an input component (**`TextInput, PasswordInput`**). In the **`handleSubmit`** function, we are using **`mutation`** instead of query as we are creating a new user instance on the backend.

Figure 9.5 shows Register route with name, email, password, confirm password fields.

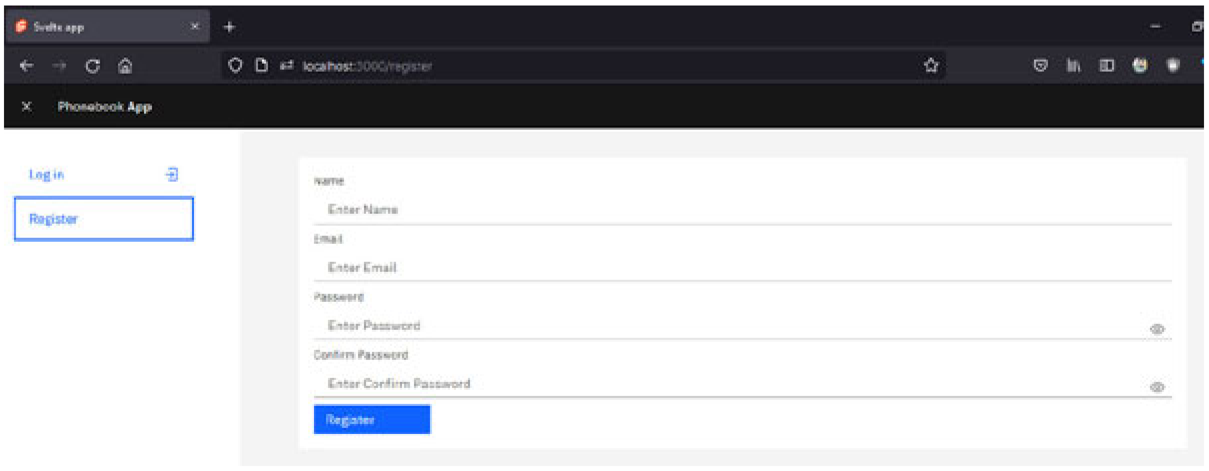


Figure 9.5: *Register Route*

Home

This will be the home page for authenticated users. It will be containing **`Link`** component that navigates to **`contacts`** path. The code for this route is as follows:

```
<script>
  import Link from “svelte-routing/Link.svelte”;
</script>

<h1>Home</h1>
<Link to=”contacts”>Contacts</Link>
```

Figure 9.6 shows Home route with **`Link`** component for **`Contacts`** route.

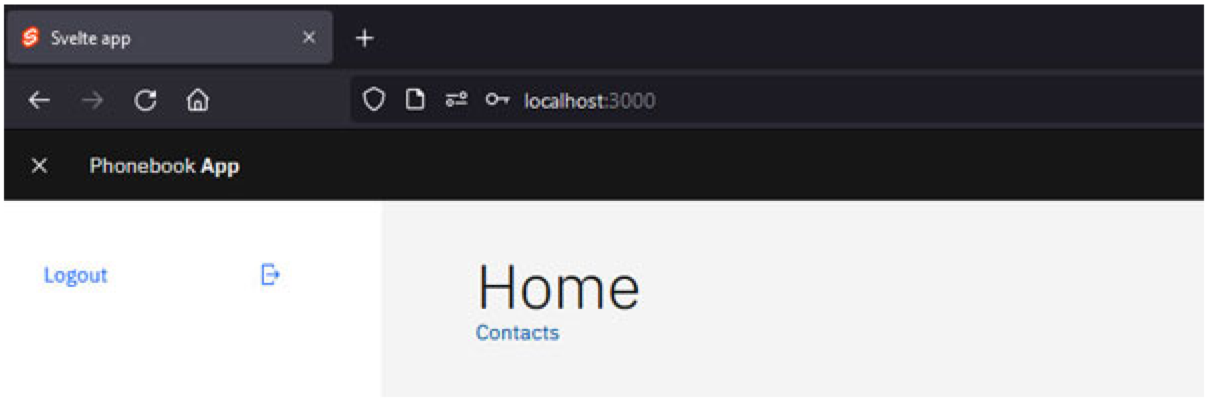


Figure 9.6: *Home Route*

NotFound

Whenever a user opens a non-existing path address, this route will get rendered. We have used Asterisk (*) for that purpose. The code for the route is as follows:

```
<script>
  import Link from “svelte-routing/Link.svelte”;
</script>

<h2>Page Not Found</h2>
Go back to <Link to=”/”>Home</Link>

<style>
  h2 {
    color: red;
  }
</style>
```

Above code is a simple page with **`Page Not Found`** message and **`Link`** to Home page.

Figure 9.7 shows **`NotFound`** component rendered for undefined path.

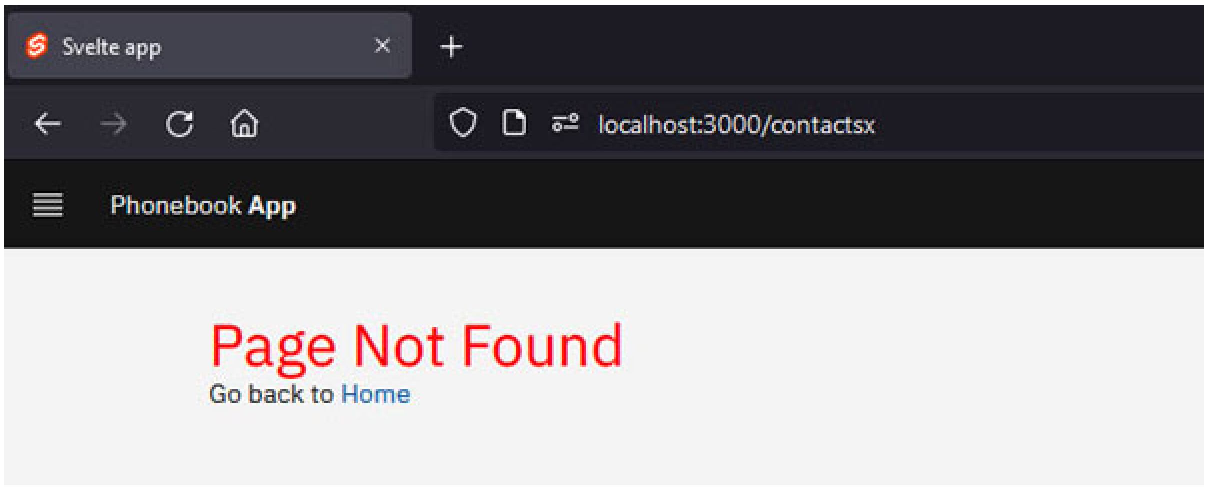


Figure 9.7: *NotFound Route*

Private route

While Login, Register are public routes (no authentication required), Home, Contacts are private routes and should only be accessible to authenticated users. We will protect these pages by using PrivateRoute (a custom route) instead of normal Route. The code for private route is as follows:

```
<script>
  import Route from “svelte-routing/Route.svelte”;
  import PrivateRouteGuard from “./PrivateRouteGuard.svelte”;

  export let path;
</script>

<Route {path}>
  <PrivateRouteGuard>
    <slot />
  </PrivateRouteGuard>
</Route>
```

In the above code we are using the **`Route`** component as wrapper. We are passing the **`path`** prop to it which is an route address. The **`slot`** element is nothing but the component to be rendered. We are wrapping it using **`PrivateRouteGuard`** component. The code for **`PrivateRouteGuard`** is as follows:

```
<script>
  import { onMount } from “svelte”;
  import { navigate } from “svelte-routing”;
  import { accessToken } from “../utils/stores.ts”;

  onMount(() => {
    console.log(“Mounting…”);
    if (!$accessToken) {
      navigate(“/login”, {
        replace: false,
      });
    }
  });
</script>

{#if $accessToken}
  <slot />
{/if}
```

`PrivateRouteGuard` uses **`onMount`** function that gets called on component mount. It checks the **`accessToken`** value in it. If **`accessToken`** is null then it will navigate to **`login`** route else it will render the child component.

Contacts

This route will be accessible to all the authenticated users and it lists all the contacts of the user. The code for contacts route is as follows:

```
<script>
  import { onMount } from “svelte”;

  import Modal from “snel-carbon/components/ComposedModal”;

  import axiod from “https://deno.land/x/axiod/mod.ts”;

  import ContactForm from “../components/ContactForm.svelte”;
  import ContactTable from “../components/ContactTable.svelte”;

  let showForm = false;
  let contacts = [];
  const accessToken = localStorage.getItem(“accessToken”);
  let editContact = null;

  const getContacts = () => {
    console.log(“fetching contacts…”);
    const query = `
        {
          contacts {
            contactId
            name
            email
            phone
            address
          }
        }
      `;
    axiod
      .post(
        “http://localhost:8080/graphql”,
        { query },
        {
          headers: { Authorization: `Bearer ${accessToken}` },
        }
      )
      .then((res) => {
        console.log(`res`, res);
        if (res.data.errors) {
          for (let error of res.data.errors) {
            alert(error.message);
          }
        }
        // console.log(`res.data.data.login`, res.data.data.login);
        if (res.data.data?.contacts) {
          contacts = res.data.data.contacts;
        }
      })
      .catch((err) => {
```

```
      console.log(`err`, err);
    });
  console.log("fetched contacts…");
};

const deleteContact = (contactId) => {
  const query = `
  mutation {
    deleteContact(contactId: "${contactId}")
  }
  `;
  axiod
    .post(
      "http://localhost:8080/graphql",
      { query },
      {
        headers: { Authorization: `Bearer ${accessToken}` },
      }
    )
    .then((res) => {
      console.log(`res`, res);
      if (res.data.errors) {
        for (let error of res.data.errors) {
          alert(error.message);
        }
      } else {
        getContacts();
      }
      if (res.data.data?.contacts) {
        contacts = res.data.data.contacts;
      }
    })
    .catch((err) => {
      console.log(`err`, err);
    });
};

onMount(async () => {
  console.log(“Mounting…”);
  await getContacts();
});

const resetForm = () => {
  showForm = !showForm;
  editContact = null;
};

const handleEdit = (event) => {
  showForm = true;
  editContact = event.detail.contact;
};
</script>

{#if showForm}
  <Modal
    bind:open={showForm}
    preventCloseOnClickOutside
```

```
      modalHeading={`${editContact ? “Edit” : “Add”} Contact`}
  >
    <ContactForm
      on:resetForm={resetForm}
      on:contactSaved={(event) => {
        console.log(`event`, event);
        showForm = event.detail.showForm;
        getContacts();
        editContact = event.detail.editContact;
      }}
      {editContact}
    />
  </Modal>
{/if}
<ContactTable
  {contacts}
  on:deleteContact={(event) => deleteContact(event.detail.contactId)}
  on:edit={handleEdit}
  on:openForm={resetForm}
/>
```

Above code renders two components namely **`ContactForm`** and **`ContactTable`**. Modal component from snel-carbon is used to show **`ContactForm`** in modal view. It defines three states **`showContact`**, **`contacts`** and **`editContact`**. **`showContact`** is a boolean flag to show or hide the **`ContactForm`** modal. contacts is an array of contacts. These contacts are fetched using **`getContacts`** function that makes network request using axiod to the GraphQL API server. It gets called once component is mounted using **`onMount`** function. We have also defined

deleteContact function that make contact delete request to the server. We have set **Authorization** header with **accessToken** for each request for user authentication purpose. **resetForm** function is used to reset the form fields on form close or submission. **handleEdit** function is used to set selected entity as **editContact**. The **editContact** data will then be passed to the form to be edited. Let’s check **ContactForm** and **ContactTable** components in detail.

ContactTable

We will show all the contacts in the table format using snel-carbon components in **ContactTable.svelte** file. We will also provide actions for **create, delete, edit** in the table. The code for the component is as follows:

```
<script>
  import { createEventDispatcher } from “svelte”;

  import DataTable from “snel-carbon/components/DataTable”;

  import Toolbar from “snel-carbon/components/DataTable/Toolbar.svelte”;
  import ToolbarContent from “snel-carbon/components/DataTable/ToolbarContent.svelte”;

  import OverflowMenu from “snel-carbon/components/OverflowMenu/OverflowMenu.svelte”;
  import OverflowMenuItem from “snel-carbon/components/OverflowMenu/OverflowMenuItem.svelte”;

  import Button from “snel-carbon/components/Button”;

  import AddAlt16 from “snel-carbon/icons/AddAlt16”;
  import Edit16 from “snel-carbon/icons/Edit16”;
  import Delete16 from “snel-carbon/icons/Delete16”;

  const dispatch = createEventDispatcher();

  export let contacts;
  $: {
    contacts = contacts.map((contact, index) => {
      contact.id = index + 1;
      return contact;
    });
  }
</script>

<DataTable
  title=”Contacts”
  sortable
  headers={[
    { key: “id”, value: “ID” },
    { key: “name”, value: “Name” },
    { key: “email”, value: “Email” },
    { key: “phone”, value: “Phone” },
    { key: “address”, value: “Address” },
    { key: “overflow”, empty: true },
  ]}
  rows={contacts}
>
  <Toolbar>
  <ToolbarContent>
    <Button
      on:click={() => {
        dispatch(“openForm”);
      }}
      icon={AddAlt16}
    />
  </ToolbarContent>
</Toolbar>

<span slot=”cell” let:cell let:row>
  {#if cell.key === “overflow”}
    <OverflowMenu flipped>
      <OverflowMenuItem
        on:click={() => {
          dispatch(“edit”, { contact: row });
        }}
        ><Edit16 />
      </OverflowMenuItem>
      <OverflowMenuItem
        danger
        on:click={() => {
          dispatch(“deleteContact”, { contactId: row.contactId });
        }}
      >
        <Delete16 />
      </OverflowMenuItem>
    </OverflowMenu>
    {:else}{cell.value}{/if}
  </span>
</DataTable>
```

In the preceding code, we are importing various components from snel-carbon like **`DataTable`**, **`Toolbar`**, **`Overflow`**, **`Button`**, and so on. We are also importing icons like **`AddAlt16`**, **`Edit16`**, **`Delete16`** from snel-carbon.

We are importing **`createEventDispatcher`** from svelte which will be used to dispatch events to the parent component. The **`contacts`** is passed as **`prop`** by parent components.

We are using **$:** (labeled statement) syntax for **reactive declarations** meaning run the code on reference value change. We are using it to add an **id** field to each contact. We are doing this because the **`DataTable`** component requires each row to have an id field. We are passing an object with **key, value** keys as **headers** to the DataTable. These will be used as header for each column. The **`contacts`** (each with **`id`**) are passed as **`rows`**. In the **`DataTable`**, we are using **`Toolbar`** with **`Add`** button that allows to add new contact.

Figure 9.8 shows ContactTable with Toolbar and headers:

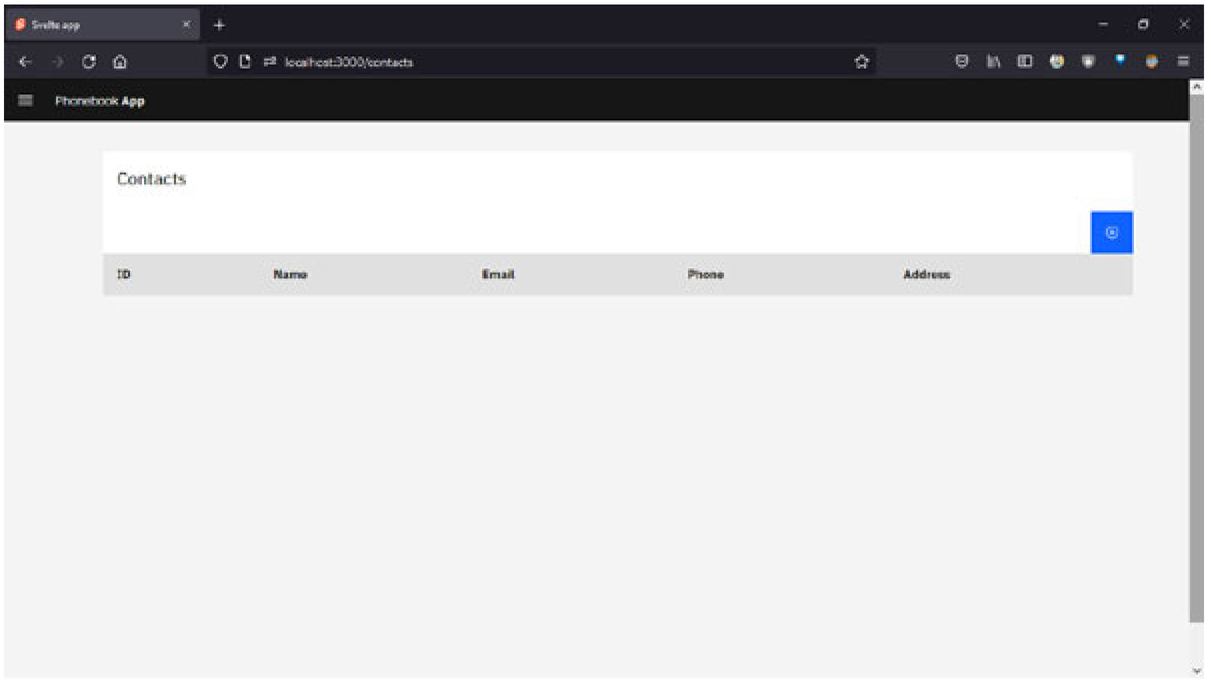
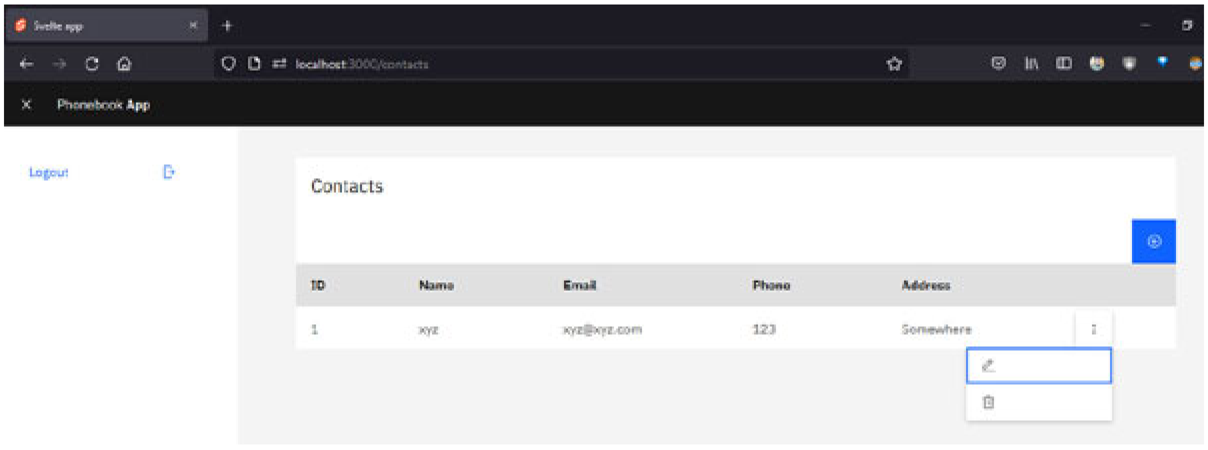


Figure 9.8: *ContactTable with toolbar*

OverflowMenu is used to to provide actions such as **`edit`**, **`delete`** the contact. We need to add { key: “overflow”, empty: true } in the **headers** to enable the overflow menu. We are using slot prop **cell to** identify the overflow cell and render it or else render the normal value for the cell. The **row** prop is passed to edit, delete actions to perform API requests using the contact data. We are dispatching these actions using **dispatch.** This means we will pass events **`edit`**, **`deleteContact`** along with state to the parent. The parent will then handle the events accordingly.

Figure 9.9 shows **`OverflowMenu`** with edit, and delete actions.



ID	Name	Email	Phone	Address
1	xyz	xyz@xyz.com	123	Somewhere

Figure 9.9: *OverflowMenu with Edit, Delete actions*

We will now check the form details used to create, edit contact on click on the **`Add`** button in next section.

ContactForm

Whenever the user clicks on the **`Add`** button, we show a modal with contact form for create or update purpose as explained in **`Contacts`** section. We will implement this component using snel-carbon components and axiod for API calls. The code for **`ContactForm.svelte`** component is as follows:

```
<script>
  import Form from “snel-carbon/components/Form”;
  import Tile from “snel-carbon/components/Tile”;
  import TextInput from “snel-carbon/components/TextInput/TextInput.svelte”;
  import TextArea from “snel-carbon/components/TextArea”;

  import Button from “snel-carbon/components/Button”;
  import InlineLoading from “snel-carbon/components/InlineLoading”;

  import axiod from “https://deno.land/x/axiod/mod.ts”;
  import { accessToken } from “../utils/stores.ts”;
  import { createEventDispatcher } from “svelte”;

  const initialContact = {
  name: “”,
  email: “”,
  phone: “”,
  address: “”,
  };

  const dispatch = createEventDispatcher();

  export let editContact;

  $: contact = editContact || initialContact;

  let loading = false;

  const handleSubmit = () => {
  loading = true;
  const addQuery = `
  mutation {
    addContact(contact: {name: “${contact.name}”, email: “${contact.email}”, phone: “${contact.phone}”, address: “${contact.address}”}){
      name
    }
  }
```

```
    `;
const updateQuery = `
mutation {
  updateContact(contact: {contactId: “${contact?.contactId}”, name: “${contact.name}”, email: “${contact.email}”, phone: “${contact.phone}”, address: “${contact.address}”})
}
`;
const query = editContact ? updateQuery : addQuery;

axiod
  .post(
    “http://localhost:8080/graphql”,
    { query },
    {
      headers: { Authorization: `Bearer ${$accessToken}` },
    }
  )
  .then((res) => {
    console.log(`res`, res);
    if (res.data.errors) {
      for (let error of res.data.errors) {
        alert(error.message);
      }
    } else {
      contact = initialContact;
      dispatch(“contactSaved”, {
        showForm: false,
            editContact: null,
          });
        }
      })
      .catch((err) => {
        console.log(`err`, err);
      })
      .finally(() => {
        loading = false;
      });
  };
</script>

<Tile>
  <Form on:submit={handleSubmit}>
    <TextInput
      bind:value={contact.name}
      required
      labelText=”Name”
      placeholder=”Enter Name”
      size=”sm”
    />
    <TextInput
      bind:value={contact.email}
      required
      type=”email”
      labelText=”Email”
      placeholder=”Enter Email”
      size=”sm”
  />
  <TextInput
    bind:value={contact.phone}
    required
    type=”number”
    labelText=”Phone”
    placeholder=”Enter Phone Number”
    size=”sm”
  />
  <TextArea
    bind:value={contact.address}
    required
    labelText=”Address”
    placeholder=”Enter Address”
    size=”sm”
  />
  {#if loading}
    <InlineLoading />
  {:else}
    <Button type=”submit” size=”small”>Save</Button>
    <Button
      on:click={() => {
        dispatch(“resetForm”);
      }}
      size=”small”
      kind=”danger”>Cancel</Button
    >
  {/if}
</Form>
```

```
</Tile>
```

In the above code, we are importing **`Tile`**, **`Form`**, **`TextInput`**, **`PasswordInput`**, **`Button`** components from snel-carbon. **`Tile`** is used as card layout for the form. **`Form`** is used to handle validations like email, number etc. and submission. **`TextInput`** is as input field for name, email (type=”email”), phone (type=”number”). **`Button`** is used to close the form and set field values to empty or with type=”submit” that validates all fields and invokes on:submit function **`handleSubmit`**.

Figure 9.10 shows **`ContactForm`** while creating new contact:

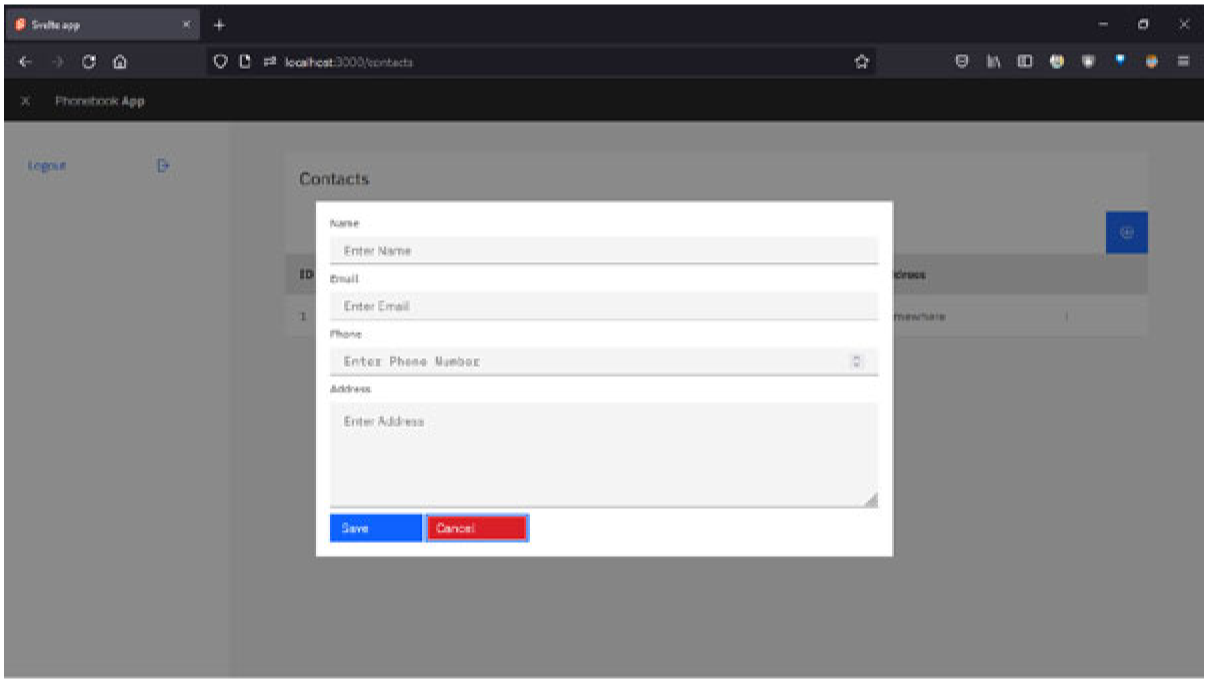
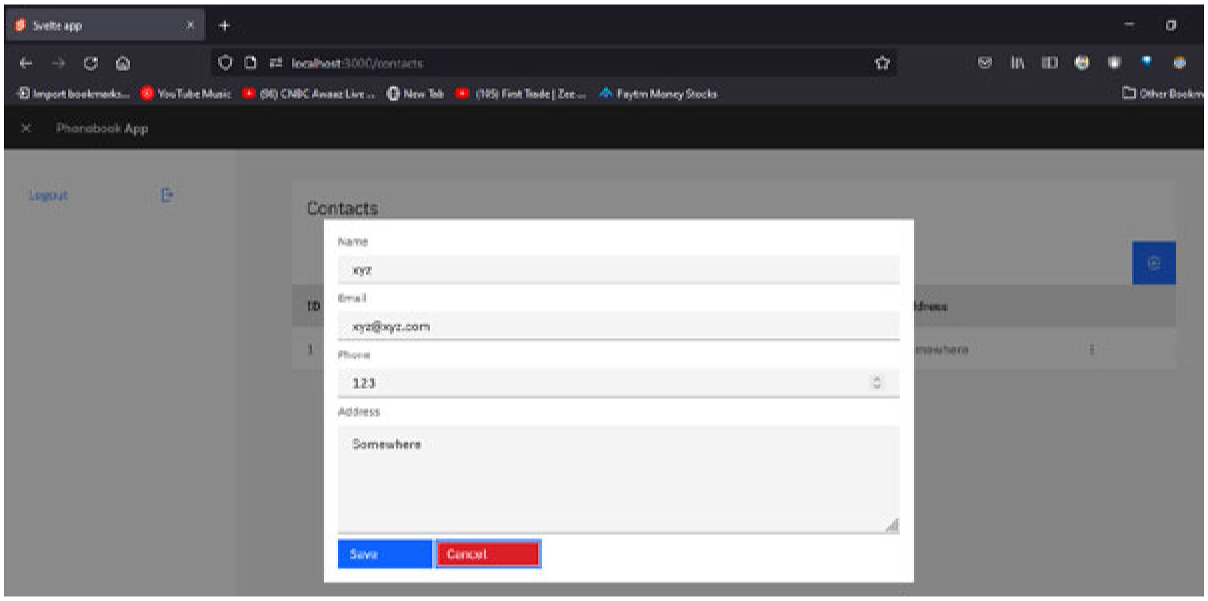


Figure 9.10: *ContactForm for new contact*

In the **script** section, we are using **`initialContact`** object to set all fields to empty on new form creation or on form close. **`editContact`** prop is used to identify whether form values should be empty for new contact using **`initialContact`** or existing form values to be used using **`editContact`**. The reactive variable **`contact`** is set according to **`editContact`** value.

The **handleSubmit** function contains two queries **addQuery** **(**create contact) and **updateQuery** (update contact)**.** Again, we are using **editContact** to select the query accordingly. Then, we are making network request for the selected query with **Authorization** header.

Figure 9.11 shows **ContactForm** with **editContact** to update the existing contact:



Name
xyz
Email
xyz@xyz.com
Phone
123
Address
Somewhere

Figure 9.11: *ContactForm for existing contact*

Layout

We will create a layout using **Header** and **Sidebar** **snel-carbon** components in this section. This will allow us to provide App title in **Header** and add login/logout buttons in the Sidebar. We will write all the layout related code in file **Layout.svelte** as follows:

```
<script>
    import Header from "snel-carbon/components/UIShell/GlobalHeader/Header.svelte";
    import SideNav from "snel-carbon/components/UIShell/SideNav/SideNav.svelte";
```

```
  import SideNavItems from “snel-carbon/components/UIShell/SideNav/SideNavItems.svelte”;

  import Authentication from “./Authentication.svelte”;

  let theme = “g10”;
  $: document.documentElement.setAttribute(“theme”, theme);

  let isSideNavOpen = false;
</script>

<Header
  company=”Phonebook”
  platformName=”App”
  bind:isSideNavOpen
  persistentHamburgerMenu={true}
/>
<SideNav bind:isOpen={isSideNavOpen}>
  <SideNavItems>
    <Authentication />
  </SideNavItems>
</SideNav>
```

In the preceding code, we are importing **`Header`**, **`SideNav`**, **`SideNavItems`** from snel carbon. The **`Header`** will be placed at top of the app while **`SideNav`** will be used to create a sidebar where **`SideNavItems`** contains actions like **`login`**, **`register`**, and so on. We have set theme to **`g10`** that is a grey colored theme. There are other themes available in the documentation.

Header

We are providing few props to the component. The **`company`** prop is used to set Title (Phonebook) for the App. The **`platformName`** (App) is a subtitle. We are the using a state **`isSideNavOpen`** to show hide the sidebar. The **`persistentHamburgerMenu`** prop is used to show hamburger button to open close the sidebar.

SideNav

We are using it along with **`SideNavItems`** to show **`Login`**, **`Register`** and **`Logout`** buttons. Our logic will be to show **`Login`** and **`Register`** buttons for unauthorized user and show **`Logout`** button to authorized user. This logic is written in **`Authentication.svelte`** file as follows:

```
<script>
  import Button from “snel-carbon/components/Button”;
  import ButtonSet from “snel-carbon/components/Button/ButtonSet.svelte”;
  import Link from “svelte-routing/Link.svelte”;
  import { accessToken } from “../utils/stores.ts”;
  import Login16 from “snel-carbon/icons/Login16”;
  import Logout16 from «snel-carbon/icons/Logout16»;
</script>

<div class=”auth-container”>
  <ButtonSet stacked>
    {#if !$accessToken}
      <Link to=”/login”>
        <Button kind=”ghost” icon={Login16}>Login</Button>
      </Link>
      <Link to="/register">
        <Button kind="ghost">Register</Button>
      </Link>
    {:else}
      <Link to="/">
        <Button
          kind="ghost"
          icon={Logout16}
          on:click={() => {
            localStorage.removeItem("accessToken");
            $accessToken = "";
          }}
        >
          Logout
        </Button>
      </Link>
    {/if}
  </ButtonSet>
</div>

<style>
  .auth-container {
    margin: 10px;
    display: flex;
    flex-direction: column;
    justify-content: space-between;
  }
</style>
```

In the preceding code, we are using carbon components **`Button`**, **`ButtonSet`** along with carbon icons **`Login16`**, **`Logout16`**. We are also importing **`Link`** component from svelte-routing and accessToken from store. The **`Link`** component is used to set url address to login, register etc. We are using **`accessToken`** to show either Login, Register buttons or Logout button. If the **`accessToken`** is valid, then we will show Logout button or else Login and Register buttons.

The **`ButtonSet`** is used to show group of buttons (Login, Register). We are passing few props to each Button component. The **`kind`** defines type of button (here, ghost). The **`icon`** prop is used to show icon in the button. We are using Login16 and Logout16 icons from snel-carbon for Login and Logout buttons respectively.

Conclusion

We have implemented a svelte application UI in this chapter using various deno modules like **`snel`**, **`snel-carbon`**, **`axiod`**, and so on. Snel module is useful for creation of Svelte applications using deno runtime. **Snel-carbon** provides components, icons based on carbon design system. We used these components, icons to create our UI. **Axiod** is used to make network requests to the server. **Svelte Routing** module is used to define various routes (public and private) for the app. We used **trex** provided by snel by default to run the scripts.

This knowledge will be helpful to set up a new deno project for Svelte UI Application. We can test the implementation by running it along with the API server from previous chapter. With this, our *Phonebook* App is completed and can be used locally or can be deployed to cloud using various platforms.

Index