```
server.route({
  method: 'GET',
  path: '/',
  handler: function (request, reply) {
    reply('Learn at http://hapibook.jjude.com!');
  }
});
```

JOSEPH JUDE

HAPI WITH TYPESCRIPT

WITH CODE SAMPLES AND DOCKER IMAGES

```
server.start((err) => {
  console.log(`Server running at: ${server.info.uri}`);
});
```

Hapi With Typescript

Build scalable applications with ease

Joseph Jude

Contents

Preface

0.1 What this book covers

Hapi.JS (called Hapi in this book) is a nodejs framework developed, originally, in Walmart Labs. What's so special about Hapi? Let us hear from the man himself who developed it. Eran Hammer developed Hapi and still serves as its lead contributor. He wrote an introductory post describing Hapi (emphasis by Eran himself):

> hapi was created around the idea that **configuration is better than code**, that **business logic must be isolated from the transport layer**, and that native node constructs like buffers and stream should be supported as first class objects. But most importantly, it was created to provide a modern, comprehensive environment in which as much of the effort is spent **delivering business value**.

As emphasised by Eran, we, the software developers, should choose tools that aid us in delivering business value as quickly as possible. Business owners want to experiment with ideas fast and move further with ideas that fetch maximum value. This is why Hapi is attractive.

Eran and team used Hapi to power the e-commerce sites at Walmart, even during the peak black friday sale. Hapi is a proven enterprise-scale nodejs framework. If you are learning a nodejs framework, you should learn the battle-tested framework.

Though originally built at Walmart labs, a large community has developed around Hapi. Hapi is now used to power e-commerce sites, fin-tech apps and so on. As you can see from Hapi community page, both large and small companies use Hapi.

In this book we will also use another great developer tool–typescript.

Over the history of JavaScript there has been many attempts to modernize JavaScript. The most recent and a successful one is an attempt by Microsoft. TypeScript was introduced in October 2012, after two years of internal development at Microsoft. TypeScript brings all the features lacking in JavaScript. Yet, it does so without abandoning JavaScript. TypeScript is just a superset of JavaScript.

So what are the benefits TypeScript brings to JavaScript?

1. TypeScript brings classes, and interfaces thereby bringing object oriented support;
2. TypeScript adds types to JavaScript, so errors are identified at compile time;
3. Because TypeScript brings static types, it has become easier to add intellisense and syntax checking to tools and IDEs;
4. Because of the same reason, it has become easier to refactor large codebase.

These features bring tremendous productivity gains for large teams with large code bases.

0.2 How this book is written

I'm writing this book in an incremental and iterative fashion, also called as agile method in the software development world. I'm using softcover as a publishing platform to distribute the book. You can read the book for free. If you want the book in an e-book format or want the accompanying code, then you have to buy the book. I'm also planning to create screencasts along with the book.

0.3 How this book is organized

Chapter 1: *Basics of Hapi & TypeScript* introduces all essential concepts of Hapi. You will learn to handle urls using routes, connect to db to store and retrieve values, and use nunjucks templates. You will get an overall view of Hapi with this chapter.

Chapter 2: *Introduction to TypeScript*, introduces all the concepts of typescript and how to compile TypeScript code automatically using `npm scripts`.

Chapter 3: *Routes, request, and reply* introduces how Hapi deals with incoming requests.

Chapter 4: *Using SQL Databases* introduces connecting to SQL dbs using TypeORM.

Chapter 5: *Hapi plugin system* introduces how to use Hapi plugin systems to modularize your Hapi project.

Chapter 6: *Validating with Joi* introduces *joi* plugin to validate route parameters.

Chapter 7: *Testing Hapi* introduces *lab* plugin to write automated test

cases.

Chapter 8: *Supporting resources* introduces hapidock, a docker based container that you can use to quickly start developing Hapi applications.

0.4 Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

A block of code is shown as below:

Listing 1: Sample Code

```
server.start((err) => {
if (err) {
        throw err;
}
console.log('Server running at:', server.info.uri);
});
```

In the course of the book, I will have to describe an ancillary topic. Instead of breaking the flow, these ancillary topics will be listed within a box as below. Same is true for important notes that I want you to remember.

Box 0.1. NOTE

Warnings or important notes appear in a box like this

Tips are also shown in boxes.

Box 0.2. TIP

Tips and tricks appear like this

0.5 What you need to follow this book

You need only 3 things besides this book: access to internet, a text editor, and a terminal application.

If you purchase the book, then you will get access to hapidock. Hapidock is a definition file to download and setup all the essential components in one go. If you do that, then you don't need internet access on an ongoing basis. Otherwise you will need internet access as you progress through the book.

There are so many choices for text editors and terminal applications in all the popular platforms. I use Vim and Visual Studio Code. Visual Studio Code comes with all batteries included. If you have not yet developed a preference, I strongly recommend Visual Studio Code.

0.6 Your feedback

I welcome your feedback. Please let me know your comments—what you liked and disliked. Your feedback will help me improve this book.

Please send your feedback to feedback@jjude.com.

0.7 Connect with me

I write regularly at my blog about software development topics. Subscribe, to learn as I write.

You can also connect with me on twitter at http://twitter.com/jjude.

0.8 Other books by me

Ionic 2: Definite Guide

Books I have written are listed at my site: https://jjude.com/books/.

0.9 Need help for your Hapi project?

- Using Hapi for your internal tools?
- A project that needs to be rescued?

You don't have to do it alone. I can help.

I can help you:

- Build your next MVP
- Testing
- Implement build and deployment automation

Get in touch with me at consulting@jjude.com.

Chapter 1

Basics of Hapi & TypeScript

This chapter gives you a comprehensive overview of Hapi. After reading this chapter, you will know:

- how to write a simple server program in Hapi and TypeScript
- how to map Hapi functions to incoming requests, generally called as routes
- how to connect to SQL db
- how to use templates

This should be sufficient enough for you to understand Hapi. Subsequent chapters will go into depth of each of these concepts.

1.1 First Hapi app

A simple server program in Hapi looks like this:

Listing 1.1: First Hapi app

```
1  const hapi = require('hapi');
2
3  const server = new hapi.Server();
4  server.connection({port: 3000});
5
6  server.start((err) => {
7    if (err) {
8      throw err;
9    }
10   console.log('Server running at:', server.info.uri);
11 });
```

This is copy of the first tutorial from Hapi site. If you run it with `node .`, it will run the server at port 3000. However, if you open your browser to `http://localhost:3000`, it will throw an error, `{"statusCode":404,"error":"Not Found"}`. That is because, this little program doesn't know how to handle the incoming requests.

Let us modify the app so that the app displays "Hello World". The additional lines are highlighted in the modified program.

Listing 1.2: Hello World in Hapi

```
1  const hapi = require('hapi');
2
3  const server = new hapi.Server();
4  server.connection({ port: 3000 });
5
6  server.route({
7    method: 'GET',
8    path: '/',
9    handler: function (request, reply) {
```

```
    reply('Hello, world!');
  }
});

server.start((err) => {
  if (err) {
    throw err;
  }
  console.log('Server running at:', server.info.uri);
});
```

This is the template for all Hapi programs.

1. `require` Hapi
2. Create a Hapi server using `server.connection`
3. Process incoming URLs via `server.route`
4. Start the server using `server.start`

1.2 Converting to TypeScript

TypeScript brings static type checking to JavaScript. Let us re-write our program in TypeScript adding types to the variables. Here is the same program in TypeScript.

Listing 1.3: Hello World in TypeScript

```
"use strict";

import * as hapi from "hapi";

const server: hapi.Server = new hapi.Server()
server.connection({ port: 3000 });
```

```
7
8  server.route({
9    method: "GET",
10   path: "/",
11   handler: (request: hapi.Request, reply: hapi.IReply) => {
12     reply("Hello World")
13   }
14 });
15
16 server.start((err) => {
17   if (err) {
18     throw err;
19   }
20   console.log("server running at 3000");
21 })
```

In line #5, we are indicating to the compiler (and to other developers who has to use our code) that **`server`** is of type **`hapi.Server`**. Similarly in line# 11, we are indicating that **`request`** is of type **`hapi.Request`** and **`reply`** is of type **`hapi.IReply`**.

This helps the IDEs as we code. The IDEs (like Visual Studio Code and Vim), bring up intellisense to aid us coding. Also the TypeScript compiler will throw up errors if we deviate from the definition of these variables.

From now on, all the programs in this book will be in TypeScript. We will learn more about TypeScript in Chapter 2.

1.3 Adding routes

When we executed our first code Listing 1.1, we got an error **`{"statusCode":404, "error":"Not Found"}`**. As we found out, that was because the code didn't have a function mapped to the incoming request. We solved that by a adding **`server.route`** method Listing 1.2.

In Hapi, **server.route** determines how a web-app should handle urls typed by a user. A **server.route** has three elements: **method**, **path**, and **handler**.

Every http request contains a method (most popular are **GET**, **POST**, and **DELETE**). You can map each of these methods with a single **server.route** or you can map multiple methods to a single **server.route**.

Path indicates the resource accessed. Path can indicate a single resource or a collection. When we type **/questions/5**, we are accessing a single question, whereas when we type **/questions**, we access all questions.

Handler function takes two parameters: **request** and **reply**. The **request** parameter contains headers, authentication information, payloads and others. The **reply** parameter is used to respond to the requests. Usually it only contains payload. But it can also have headers, content types, content length and so on.

Let us put all of these into a code snippet.

Listing 1.4: Handling routes

```
server.route({
  method: "GET",
  path: "/",
  handler: (request: hapi.Request, reply: hapi.IReply) => {
    reply("This is a GET method")
  }
});

server.route({
  method: ["POST","PUT"],
  path: "/new",
  handler: (request: hapi.Request, reply: hapi.IReply) => {
    reply("This is a " + request.method + " method")
  }
});

server.route({
  method: "GET",
```

```
19    path: "/questions/{id}",
20    handler: (request: hapi.Request, reply: hapi.IReply) => {
21      reply("Question requested is: " + request.params.id);
22    }
23  });
```

There are three blocks of `server.route` here.

The first code-block is a simple one. Here, we handle only one http method, `GET`. It serves root path `/` and `reply` with a message.

The second code-block handles both `POST` and `PUT` http methods. It serves the urls at path `/new`. It just prints type of incoming http method with `request.method`. You can use `request.method` to decide the logic if some part of the code has to be different.

The third code-block serves a `GET` url with a parameter, `id`. The parameter is sent back as a reply.

We will learn more about routes in Chapter 3.

1.4 Using plugins

Hapi is designed in a modularized manner. It has all the essential features to build server applications. Everything else is modularized into plugins.

Hapi has a plugin for templating, input validation, authentication, testing and so on. We will learn of some of these plugins. You can view all the available Hapi plugins in the Hapi website. Some of these plugins are official plugins created by Hapi team. There are also plugins created by the Hapi community.

You have to register a plugin before using it. You register a plugin as

below:

```
server.register(require("plugin"));
```

We will learn more about plugins in Chapter 5.

1.5 Connecting to DB

When it comes to databases, Nodejs is often associated with NoSQL dbs like Mongo. Yet, SQL dbs like MySQL, Postgresql have been battle tested in real world scenarios for years.

In this book, we'll learn how to connect to db from Hapi using TypeORM, a data mapper based ORM, to connect to db. Specifically, we will connect to Postgresql. The same concept can be used to connect to any relational db.

In Chapter 4, we will see how to install TypeORM. In this section, let us see how to use TypeORM to connect to db.

Using db involves, connecting to it, defining models (tables), writing into it, and reading from it. Let us see each of these in this section.

Listing 1.5: Connection parameters to connect to postgres db

```
1 import { createConnection } from "typeorm";
2
3 createConnection({
4   driver: {
5     type: "postgres",
6     host: "localhost",
7     username: "pg_user",
8     password: "pg_password",
9     database: "pg_dbname",
```

```
    port: 5432
  }
}).then(connection => {
    console.log("db connected");
  })
```

1.5.1 Defining models

In TypeORM, `Entity` decorator defines a table, `Column` decorator defines columns, and `PrimaryGeneratedColumn` defines a auto generated primary column. Here is how you will define a blog post entity.

Listing 1.6: Defining models in TypeORM

```
import { Entity, PrimaryGeneratedColumn, Column } from "typeorm";

@Entity()
  export class Entry {

    @PrimaryGeneratedColumn()
      id: number;

    @Column()
      title: string;

    @Column()
      content: string;
  };
```

TypeORM has to create or modify these tables in the db. For that purpose, the defined models have to be part of the connection parameters. The additional lines to include the defined models are highlighted.

Listing 1.7: Including models in connection parameters

```
createConnection({
  driver: {
```

```
    type: "postgres",
    host: "pg",
    username: "postgres",
    password: "secret",
    database: "postgres",
    port: 5432
  },
  entities: [
    Entry
  ]
}).then(connection => {
  console.log("db connected and models created");
})
```

1.5.2 Reading and writing entities

TypeORM provides **`entityManager`** and **`repository`** to deal with entities. TypeORM documentation recommends using **`repositories`** to connect to entities. So we will use **`repositories`**. The following snippet will fetch the corresponding repository and fetch all entries in the table.

Listing 1.8: Fetching all entries

```
let entryRepo = connection.getRepository(Entry);
let allEntries = await entryRepo.find();
```

Now we can loop through **`allEntries`** and pick up individual **`entry`** for processing. Similarly, **`entryRepo.persist(newEntry)`** persists a new entity.

1.5.3 Putting it all together

Now let us put the entire program together. In the following snippet, we define two routes: a GET route that fetches all entries, and a POST route that creates a new entry. We are using **async**, **await** for cleaner code.

Listing 1.9: Using TypeORM to connect to db

```
"use strict";

import * as hapi from "hapi";
import "reflect-metadata";
import {createConnection, Entity, PrimaryGeneratedColumn, Column} from "typeorm";

const server: hapi.Server = new hapi.Server()
server.connection({ port: 3000 });

@Entity("Entries")
export class Entry {

  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  title: string;

  @Column()
  slug: string;

  @Column()
  content: string;
};

server.route({
  method: "GET",
  path: "/",
  handler: async (request: hapi.Request, reply: hapi.IReply) => {
    const entryRepo = server.app.dbConnection.getRepository(Entry);
    const entries = await entryRepo.find();
    reply(entries);
  }
});

```

```
server.route({
  method: "POST",
  path: "/",
  handler: async (request: hapi.Request, reply: hapi.IReply) => {

    let {title,slug,content} = request.payload;

    let newEntry = new Entry();
    newEntry.title = title;
    newEntry.slug = slug;
    newEntry.content= content;

    let entryRepo = server.app.dbConnection.getRepository(Entry);
    await entryRepo.persist(newEntry);

    reply("New post created");
  }
});

createConnection({
  driver: {
    type: "postgres",
    host: "pg",
    username: "postgres",
    password: "secret",
    database: "postgres",
    port: 5432
  },
  entities: [
    Entry
  ]
}).then(async connection => {
  server.start(err => {
    if (err) {
      throw err;
    }
        // set the connection to server.app so it can be used in methods
        server.app.dbConnection = connection;
    console.log("server running in " + server.info.uri);
  })
})
.catch(err => console.log(err));
```

Here all activities, defining models, connecting to db, defining routes, and bringing up the server, all happens within the same file. This is not

a recommended practice. In the coming chapters, we will modularize them.

1.6 Summary

- Hapi is a nodejs framework to build web applications and back-end servers;
- Hapi is a modularized framework with rich set of plugins;
- TypeScript brings static type checking to JavaScript;
- Create a Hapi server with `server.connection` and `server.start`;
- Use `server.route` to map incoming urls to handling functions;
- `server.route` has three components: `method`, `path`, and `handler`;
- `vision` is the plugin to connect templates to Hapi
- We can use TypeORM to connect to SQL db;

Chapter 2

Introducing TypeScript

In this chapter you will learn about TypeScript, a Javascript compiler from Microsoft. After reading this chapter, you will know:

- the problems TypeScript solves
- how to use TypeScript
- basics of TypeScript
- compiling TypeScript automatically using npm scripts

2.1 Why TypeScript?

Real world applications take months, if not years, to build. As we build, we add new blocks of code, refactor existing code, and add and remove 3rd party libraries. As we change our application, we need early feedback about the impact of these changes. One such feedback is about the type of the variables we use. When we develop in a typed language like

C, Swift, or Java, the compiler (and the code editor) provides these early feedback. Javascript lacks this feature.

Let us take an example. Say we are developing an address-book application and we want to define a contact. Our definition might look like this, in JavaScript.

```
var name, age, city;
```

If another developer were to use these variables, they won't know if age is an integer or a string. You might have *intended* it to be a string, while the other developer might use it as an integer. Now the application breaks at the **run time**. Too late!

This is where TypeScript helps. In TypeScript, the same definition will look like this:

```
let name: string,
    age: number,
    city: string;
```

Here type of the variable is defined after `:`. Now it is clear for both the compiler and for anyone reading the code. If another member of the team were to make the same mistake as assigning string value to age, the code editor and the compiler would identify and display the error. They can change the code at the compile time instead of wasting hours debugging later. Think of all the 3rd party libraries we use in our applications. Wouldn't such type system make a huge difference in our productivity?

This is not the only benefit TypeScript brings.

TypeScript supports the evolving JavaScript features, like async and decorators. TypeScript also brings object-oriented programming (OOP)

features like classes and interfaces to JavaScript. All of these are transpiled into classical JavaScript.

2.2 Components of TypeScript

TypeScript is both a **type checker** and a **transpiler**. As a type checker, TypeScript, verifies your code against the defined type annotations. If it finds any mismatch, it shows an error. As a transpiler, it compiles TypeScript code into JavaScript code. As a transpiler, it strips all type definitions and generates classical JavaScript. If you use any evolving features, then again TypeScript transpiles them into classical JavaScript version.

2.3 Installing and using TypeScript

You can install TypeScript using `npm` (assuming you have already installed node and npm).

Listing 2.1: Installing TypeScript

```
npm install -g typescript
```

Now you can invoke TypeScript from the command-line with `tsc <filename>`.

Let us start with a simple program. Let us continue with the person in an address-book example we started with. In TypeScript it will be coded as in Listing 2.2.

Listing 2.2: First TypeScript Program

```
class Person {
  public Name: string;
  public Age: number;
  public City: string;

  constructor(name: string, age: number, city: string) {
    this.Name = name;
    this.Age = age;
    this.City = city;
  }
}

let singer = new Person("Elvis Presley", 42, "Memphis");
console.log(singer.Name);
```

You can compile this with `tsc person.ts`. TypeScript will produce the following JavaScript code.

Listing 2.3: Transpiled Javascript Program

```
var Person = (function () {
  function Person(name, age, city) {
    this.Name = name;
    this.Age = age;
    this.City = city;
  }
  return Person;
}());
var singer = new Person("Elvis Presley", 42, "Memphis");
console.log(singer.Name);
```

You can execute this transpiled code as like any other JavaScript code, with `node person.js`. It will output Elvis Presley.

Now let us see how TypeScript helps as we code. I'm using Visual Studio Code to illustrate intellisense. But this works across other text editors that support TypeScript.

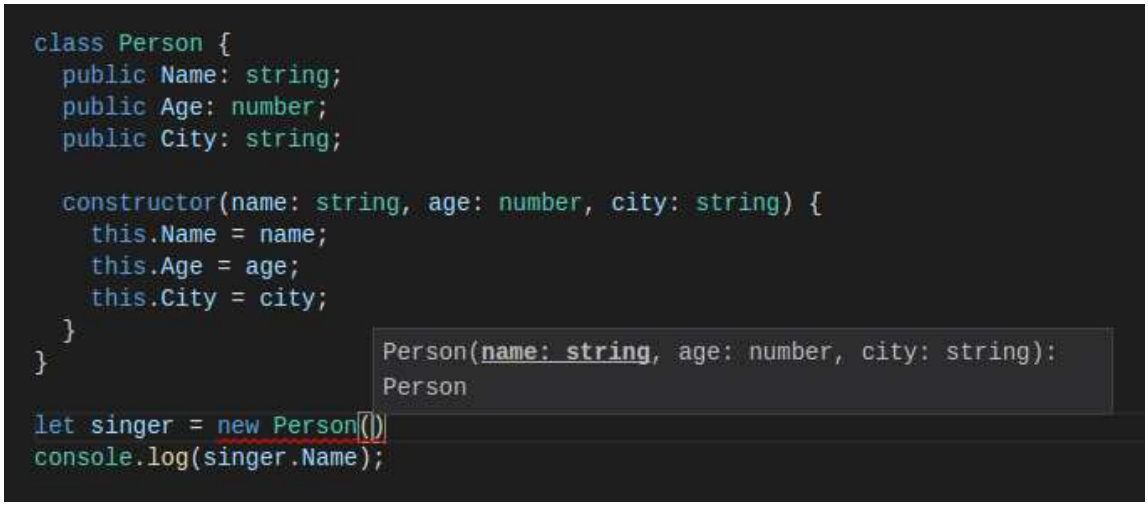

Figure 2.1: Intellisense in VS Code

As you can see from Figure 2.1, the code editor, in this case VS Code, shows the definition of the class as you type. This aids not only helps us to be productive, but also reduces errors.

What if you mistakenly assign a wrong type? Again the TypeScript type checker comes to help. It will show the error with a squiggly line as in Figure 2.2.

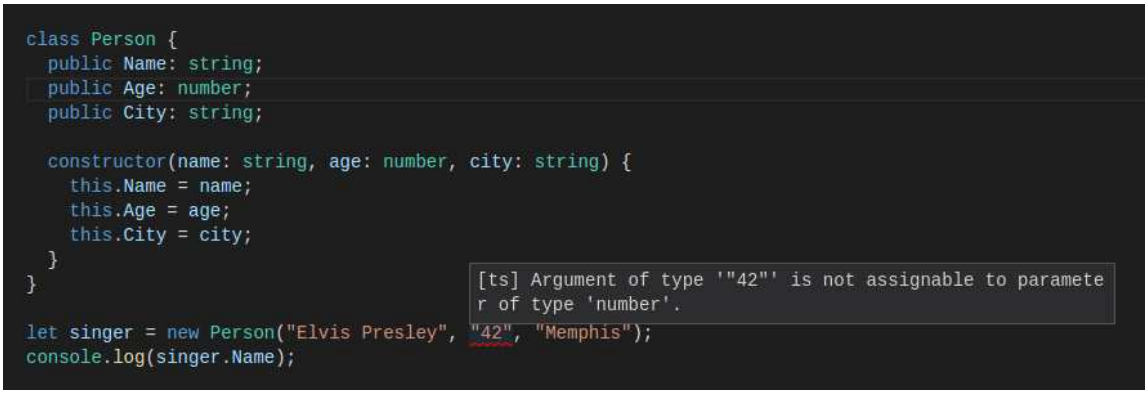

Figure 2.2: Type error shown in VS Code

If you ignore and continue, TypeScript compiler will show the error when you compile the program. Say you go ahead and compile the above program, with the error. You will see an error as shown in Listing 2.4.

Listing 2.4: TypeScript showing type error at compile time

```
$ tsc person.ts
person.ts(13,42): error TS2345: Argument of type '"42"' is not
assignable to parameter of type 'number'.
```

TypeScript provides all the help it can to avoid errors. It won't force you to modify though. It will still compile into JavaScript.

2.4 Types for existing modules

TypeScript is getting popular and developers are writing new JavaScript modules in TypeScript. TypeORM, which we will learn in this book, is one such example. But there are other modules which are not written in TypeScript, like Node and Hapi. How to use TypeScript with them?

To use TypeScript with existing JavaScript modules, we need their Type definitions. Think of type definitions as bridge between TypeScript and the module written in JavaScript.

Listing 2.5: Installing TypeScript dependencies

```
npm install @types/node @types/hapi --save-dev
```

This will install type definitions for `node` and `hapi` as a

`devDependencies`. Now you can use them within a TypeScript project, as if they were written in TypeScript.

Box 2.1. Module Dependencies in Node.js

When you develop an application, you build upon other modules. Even when you are building a simple `hello world` application in Hapi, you are dependent on the `hapi` package. While developing your application, these dependencies are installed using `npm install`. When you install these modules an entry is marked in `package.json`.

This way `npm` knows the exact modules to install when you migrate to other environments like test, staging, and production. When your team memebrs install the application, they can install all the required modules.

Certain modules are required only during development, like type definitions and test related modules. These modules are stored under `devDependencies`. The modules listed under `devDependencies` are not installed in production mode, i.e, when the environment is set using `NODE_ENV` or when you install via `npm install -production`.

2.5 TypeScript Basics

As a superset of JavaScript, TypeScript builds on existing JavaScript features. It supports comments, data types, functions, and operators. Here in this section, let us understand the elements TypeScript adds to JavaScript.

2.5.1 Comments

As a superset of JavaScript, TypeScript supports both single line and multiline comments.

Listing 2.6: Comments in TypeScript

```
// this is a single line comment
/* this is a multi line comment.
   Multiline comments starts with slash-star and ends with star-slash */
```

2.5.2 Declaration

In TypeScript you use `const` to declare a constant and `let` to declare a variable.

Listing 2.7: Variables declaration in TypeScript

```
let  name = "TypeScript";
const pi = 3.141;
```

TypeScript infers type from declaration. In the above example, `name` is a variable and `pi` is a constant. Once an identifier has a type, that type can't be changed.

You can also specify the type by writing it after the variable, separated by a colon.

Listing 2.8: Explicit type declaration in TypeScript

```
let name: string = "TypeScript";
```

2.5.3 Basic Types

TypeScript supports all JavaScript types. Here are the basic types.

- **Boolean**: a true of false value.
- **Number**: floating point value, as in JavaScript.
- **String**: textual data, defined with single or double quotes. Multi-line strings are defined with backtick(‘).
- **Any**: any type. Used when impossible to know the type.
- **Void**: absence of type. Used to indicate that a function does not return a value.

Listing 2.9: TypeScript Basic Types

```
// boolean
let isDone: boolean = false;

// number
let loc: number = 600;

// string
let name = "typescript";
name = 'javascript'

// multi-line string
let subject = `TypeScript is awesome.
Google adapting a language developed by Microsoft, shows how awesome it is.
`

// any
let userInput: any;
userInput = "any user input";
userInput = 45.3;
userInput = false;

// void
```

```
function showWarning(): void {
  alert("This is a warning");
}
```

2.5.4 Collections

Collections is grouping of multiple elements into a single unit. Here are the collection types TypeScript supports.

- **Array**: Arrays could be typed or generic
- **Tuple**: Used to group fixed number of elements. Their type need not be the same.
- **Enum**: Gives friendly names to sets of numeric values. By default, enums begin numbering their members starting at 0. You can override this manually. You can set values for all elements manually too.
- **Union**: Variable of multiple types.

Listing 2.10: TypeScript Collections

```
// array
let cities: string[] = ['delhi', 'chennai', 'mumbai'];
let cities: Array<string> = ['delhi', 'chennai', 'mumbai'];

// tuple
let yearBorn: [string, number];
yearBorn = ['julia roberts', 1967];

// enum
enum Day {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
let firstDay: Day = Day.Sunday;
```

```
// override or set values manually
enum Day {Sunday = 1, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
enum Direction {North = 2, South = 4, East = 6, West = 8};
enum Direction {North = 2, South = 4, East, West}; // East = 5; West = 6

// union
let path: string[] | string;
path = ['/home', '/home/dropbox'];
path = '/home';
```

2.5.5 Type Assertions

Sometimes you would want to override the inferred type. Then you can use type assertions. There are two forms of type assertion. One is using "angle-bracket", and another is using "as". Type assertions are different from type casting found in other language. Casting is a runtime operation, whereas assertion is a compile time operation.

Listing 2.11: Type Assertions

```
// using <>
let name: any = "Bruce Wills";
let nameLen: number = (<string>name).length;

// using as
let name: any = "Bruce Wills";
let nameLen: number = (name as string).length;
```

2.5.6 Control Flow

In this section, we will see decision-making and looping statements.

if . . . else

The `if-else` statement is a decision-making statement. The syntax of the statement is:

Listing 2.12: If-Else

```
if (condition) {
  statement1
}
else {
  statement2
}
```

`else` part of the statement is optional. You can also check for multiple conditions with `if..else if..`.

Listing 2.13: If..Else..If

```
if (condition1) {
  statement1
} else if (condition2) {
  statement2
} else {
  statement3;
}
```

ternary operator

This is a simplified, concise `if-else` statement. It takes the form: `boolean-expression ? statement1 : statement2`.

Listing 2.14: Ternary Operator

```
let speed = 80;
let isFast = speed > 55 ? true : false
```

for loop

Use **for** loop when you know how many times a task has to be repeated.

Listing 2.15: For loop

```
let sum = 0;

for (let i = 0; i<= 1000; i++){
  if (i % 3 == 0 || i % 5 == 0){
    sum = sum + i;
  }
}

console.log(sum);
```

for-of loop

TypeScript introduces a **for-of** loop (since TypeScript is a superset, you can also use the existing For-in loop) to loop through collections.

Listing 2.16: For..of loop

```
let cities: string[] = ['delhi', 'chennai', 'mumbai'];

for (let city of cities) {
  alert(city);
}
```

switch

The **switch** statement is an enhanced **if-else** statement, which is convenient to use if there are many options to choose.

Listing 2.17: Switch statement

```
let animal = 'dog';

switch (animal){
  case 'dog':
    alert('dog');
    break;
  case 'cat':
    alert('cat');
    break;
  default:
    alert('none?')
}
```

while

Use **`while`** to execute a task until a given condition is true.

Listing 2.18: While loop

```
let sum = 0;
while (sum <= 5) {
  sum = sum + 1;
}
```

do...while

`do...while` is similar to **`while`** loop, except that the statement is guaranteed to run at least once.

Listing 2.19: Do..while

```
let sum = 0;
do {
  sum = sum + 1;
} while (sum <= 4)
```

2.5.7 Functions

In TypeScript you declare functions similar to JavaScript, with type information about the input parameters and return type.

Listing 2.20: Functions

```
function squareOf(i: number): number {
  return i * i;
};
```

Return types can be interfered, so we can declare a function like this:

Listing 2.21: Infer return type

```
function squareOf(i: number) { return i * i };
```

In TypeScript, as in JavaScript, functions are first-class citizens. This means we can assign functions to variables, and pass functions as parameters. We can also write anonymous functions. All of these will generate the same JavaScript.

Listing 2.22: Anonymous Functions

```
// function as a variable
let sqr1 = function sqr (i: number) : number {
  return i * i;
}

// anonymous function
let sqr2 = function (i: number) : number {
  return i * i;
}

// alternate syntax for anonymous function using =>
let sqr3 = (i: number) : number => { return i * i;}
```

```
// return type can be inferred
let sqr4 = (i: number)  => { return i * i;}

// return is optional in one line functions
let sqr5 = (i: number) => i * i
```

Optional and default values

Functions can take optional values. You mention the optional values by using `?:` syntax.

Listing 2.23: Optional values for function parameters

```
function getFullName(firstName: string, lastName?: string) : string {
  if (lastName) {
    return firstName + " " + lastName;
  } else {
    return firstName
  }
}
```

You can also mention default values for parameters.

Listing 2.24: Default values for function parameters

```
function getFullName(firstName: string, lastName: string = "") : string {
  return (firstName + " " + lastName).trim();
}
```

2.5.8 Classes

TypeScript brings object-oriented approach to JavaScript. Let us consider an example.

Say you are developing a digital library of books. Then you can define a Book class like this:

Listing 2.25: Defining class

```
class Book {
  name: string;
  purchasedYear: number;

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

let Book1 = new Book('7 habits', 2005);
```

TypeScript also supports sub-classing or inheritance. If you want to extend the digital library to include all assets like CDs, PDFs and so on, you can modify the above class into a super-class and many sub-classes.

Listing 2.26: Class Inheritance

```
class Asset {
  name: string;
  purchasedYear: number;

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

class Book extends Asset {
  constructor (name: string, purchasedYear: number) {
    super(name, purchasedYear);
  }
}

let book1 = new Book('7-habits', 2013);
```

2.5.9 Interfaces

Interfaces allow multiple objects to expose **common** functionality. By using interface, you can enforce that all these assets implement a common functionality, say name, purchased year, and age. Interface is only a contract, implementation is carried out at the classes level.

Listing 2.27: Interfaces

```
interface iAsset {
  name: string;
  purchasedYear: number;
  age: () => number;
}

class Book implements iAsset{
  name: string;
  purchasedYear: number;

  constructor (name: string, purchasedYear: number){
    this.name = name;
    this.purchasedYear = purchasedYear;
  }
  age() {
    return (2016 - this.purchasedYear);
  }
}

let Book1 = new Book('7 habits', 2005);
console.log(Book1.age());
```

2.5.10 Decorators

Decorators is one of the experimental features that TypeScript is bringing to Javascript tool-chain. We will not create decorators in this book, but will use it while using ORM in Chapter 4. So, this section explains only about using decorators.

Decorators modify a class, property, method, or method parameter. They provide metadata and specify extra behaviour. The syntax for using decorator is with "@". Let us take an example to understand this.

We started with a Person class in Listing 2.2. Say someone else created a decorator (remember we are not dealing with creation of a decorator in this book), that prints a message every time a new instance is created. Then we would decorate the class as in Listing 2.28

Listing 2.28: Decorated Person class

```
@logClass
class Person {
  public Name: string;
  public Age: number;
  public City: string;

  constructor(name: string, age: number, city: string) {
    this.Name = name;
    this.Age = age;
    this.City = city;
  }
}
```

Now every time there is a new Person created, it will print a message.

The same can be extended to properties and methods. We will see more in Chapter 4 as we define models and columns.

2.5.11 Async, await

Javascript, and nodejs, is single-threaded. This works fine until you have to call asynchronous tasks like writing to a db or calling a web-service. We handle these async functions using callbacks and promises. TypeScript introduces async..await to write these async functions, much the same way as you write a sync function.

Listing 2.29 is a long-running, time-consuming process like a web-service or writing to a db Listing 2.30 invokes this long-running process using the new async-await syntax.

Listing 2.29: A time-consuming function

```
function delay(ms: number) {
  return new Promise<void>(resolve => {
    setTimeout(resolve, ms);
  });
}
```

Listing 2.30: Calling with async..await

```
async function asyncLog(){
  console.log("starting");

  await delay(500);
  console.log("after 500ms");

  await delay(1000);
  console.log("after 1000ms");
}
```

If you invoke this function, **`asyncLog`**, the output will be like this:

```
starting
// after 500 ms
after 500 ms
//after 1000 ms
after 1000 ms
```

The code becomes lot less verbose by using async..await. We don't have to have lot of callbacks or **`then`** functions.

2.6 Automating your workflow with npm scripts

Even though TypeScript brings ton of benefits, you can't execute TypeScript files directly. You need to compile it into Javascript file, which then has to be executed. You compile with the command `tsc <filename>`. This compilation has to be repeated every time the file is changed. This is a boring task, that could be automated.

Many people use task runners like `gulp`. There is no harm in using an additional module. But in this section, we are going to use `npm` scripts to automate compiling TypeScript files.

Why use npm instead of a task runner?

- we're already using npm while using node
- it reduces number of dependencies. This reduces both total project size (as no more gulp modules are needed), and number of things that can go wrong.

Let me start with `package.json`. This is the default `package.json` when you start with `npm init`.

Listing 2.31: Default package.json

```
{
  "name": "new project",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

With `npm run` you can execute scripts in the `scripts` block. If you run it now, you'll get an error:

```
Lifecycle scripts included in new:
test
echo "Error: no test specified" && exit 1
```

This is expected. Why? There is only one script mentioned, and that echos that there is no test. We will learn about testing in Chapter **??**. For now, let us add scripts specific to TypeScript.

First is a script to **compile TypeScript** files. This script below invokes typescript compiler.

Listing 2.32: npm script to compile TypeScript

```
"scripts": {
  "compile": "tsc --outDir ./build --module commonjs ./src/*.ts"
}
```

Now if you invoke, `npm compile`, typescript files in `./src` folder will be compiled into `./build` folder.

Can you start the node server automatically when the compilation succeeds? Yes, you can. The compile command becomes:

Listing 2.33: Starting node server after compilation

```
"compile": "tsc --outDir ./build --module commonjs ./src/*.ts && \
node ./build/server.js"
```

This is fine. But whenever you change the file, you need to run `npm compile`. Can it watch for changes to files and invoke this command automatically?

Yes it can.

Nodemon provides the watching facility. Nodemon can watch for changes for certain file extensions and execute a command. So let us add another script to the scripts block.

Listing 2.34: Watching files for change

```
"scripts": {
  "start": "./node_modules/nodemon/bin/nodemon.js -e ts  --exec \"npm run compile\"",
    "compile": "tsc --outDir ./build --module commonjs ./src/*.ts && node ./build/server.js"
}
```

If any **ts** files changes, then nodemon will invoke **npm compile**, which in turn compiles and (re-)starts the node server.

Now you can start the server with **npm start** and go on with your development. Whenever you change a TypeScript file, it will be automatically compiled and restart the server.

2.7 Summary

- TypeScript adds **type checker** and **intellisense** to nodejs development tool chain.
- TypeScript is a superset of JavaScript.
- TypeScript compiles into classical JavaScript code.
- Type definitions are available for existing JavaScript modules.
- TypeScript adds planned features of JavaScript, like decorators and async, for today's development.

- TypeScript brings OOPs concepts to JavaScript
- TypeScript can be compiled with **`npm scripts`**.

Chapter 3

Routes, request, and reply

The primary responsibility of a web-application server is to process the incoming URL and send back a response. Hapi uses a concept called *routing* to map the incoming url to a processing function. In this chapter, you will learn:

- how Hapi maps incoming URLs to a processing function;
- elements of this routing mechanism;
- how to process parameters in the URL;
- how to code a catch-all routing function;
- how to organize routes;

3.1 Basics of routing

Typical workflow of a web-application is depicted in Figure 3.1.

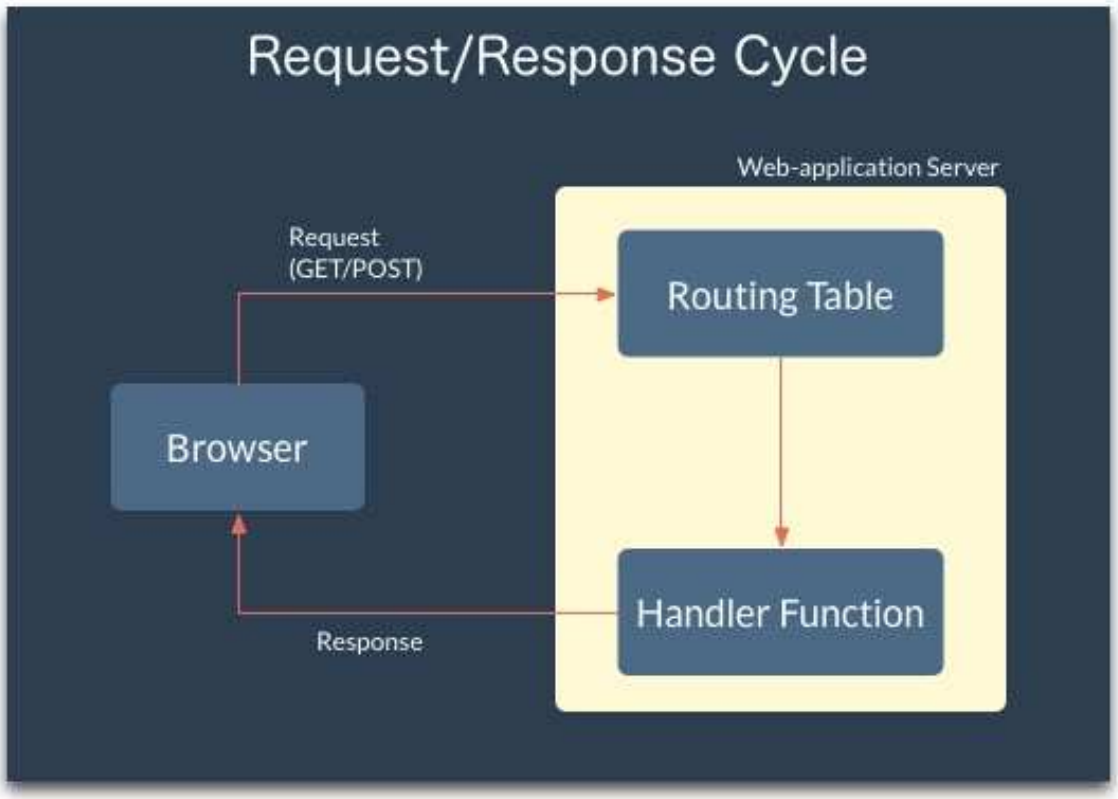

Figure 3.1: Request Response Cycle of a Web-Application

- User types an URL in the browser;
- The web-application server gets that URL and according to a pre-defined algorithm, finds a function to forward the URL;
- That function then processes the URL and sends back a response to the browser.

In Hapi, `server.route` does this job of routing the incoming request to a processing function.

Let us start with the basic `hello world` program we wrote in Chapter 1. Pay attention to the highlighted lines of code.

Listing 3.1: Hello World in Hapi

```
"use strict";

import * as hapi from "hapi";

const server: hapi.Server = new hapi.Server()
server.connection({ port: 3000 });

server.route({
  method: "GET",
  path: "/",
  handler: (request: hapi.Request, reply: hapi.IReply) => {
    reply("Hello World")
    }
});

server.start((err) => {
  if (err) {
    throw err;
  }
  console.log("server running at 3000");
)
```

Here, `server.route` maps the handler function to a `GET` request at the path `/`. The handler function replies with "Hello World". It could be a json or a html response.

There are three elements we see in `server.route`: `method`, `path`, and `handler`. In this section, we will understand these three elements in detail.

3.2 Route Methods

Every http request contains a method. It will be one of `GET`, `POST`, `PUT`, or `DELETE`. There are other methods too, but for our discussion, we will limit to these methods. The methods are:

- **GET**: retrieves a resource or a collection of resources. Ex: `GET /users` or `GET /users/1`;
- **POST**: create a new resource. Ex: `POST /users`;
- **PUT**: update a resource. Ex: `PUT /users/1`;
- **DELETE**: delete a resource. Ex: `DELETE /users/1`;

Let us see how we can handle these methods in Hapi.

Listing 3.2: One route function per method

```
server.route({
  method: "GET",
  path: "/",
  handler: (request: hapi.Request, reply: hapi.IReply) => {
    reply("This is a GET method");
  }

server.route({
  method: "POST",
  path: "/",
  handler: (request: hapi.Request, reply: hapi.IReply) => {
    reply("This is a POST method");
  }
```

Here, we are handling each http method in its own `server.route` method. This is easy to read and understand.

We can also handle multiple http methods in a single route.

Listing 3.3: Multiple methods in a single route funtion

```
server.route({
  method: ["GET","POST"],
  path: "/",
  handler: (request: hapi.Request, reply: hapi.IReply) => {
    reply("Got " + request.method + " method");
  }
```

This is handy to handle a forms on the front-end. We can display the form with `GET` and handle the input with `POST` using the same handler.

3.3 Path Parameters

Have you noticed the URLs when you access Stackoverflow site? Let us consider some of the URLs from Stackoverflow.

- http://stackoverflow.com/questions/tagged/hapijs lists the questions tagged Hapijs
- http://stackoverflow.com/q/26243489/ is a specific question on Hapijs
- http://stackoverflow.com/a/26261428 is the first answer to that question

What do we find from these URLs?

1. Tagged questions can be accessed through URLs of the pattern: `http://stackoverflow.com/questions/tagged/<tag>`. Here `<tag>` will vary. It could be hapijs, nodejs, python and so on. This is the variable parameter in this URL.
2. Questions are accessed through the urls of the pattern: `http://stackoverflow.com/q/<questionId>`. `q` indicates a questions, and an id indicates the question id to display. Here the variable parameter is questionId.
3. Answers are accessed through the URLs of the pattern: `http://stackoverflow/a/<answerId>`. As above, `a` indicates it

is an answer, and an id indicates the answer to display. Here the variable paramers is answerId.

If we were developing Stackoverflow, we have to name these variable parameters, so that we can identify the specific resource that we need to send back. In Hapi, we name the variables between `{}`. So our variables could be `{tag}`, `{questionId}`, and `{answerId}`. These variables are accessed within the handler function as part of `request.params`. So these variables might be accessed as, `request.params.tag`, `request.params.ques` and `request.params.answerId`.

Listing 3.4: Accessing Path Parameters

```
server.route({
  method: "GET",
  path: "/questions/{id}",
  handler: (request: hapi.Request, reply: hapi.IReply) => {
    reply("Question requested is: " + request.params.id);
    }
});
```

Hapi also supports multiple parameters in a path. Each parameter can be accessed with its name. If you use twitter and their lists functionality, then you would've noticed multiple parameters in path. I have many twitter lists and one of them is a "tech" list. The path to it is `https://twitter.com/jjude/lists/tech`. Here `jjude` is username parameter and `tech` is listname parameter. Listing 3.5 shows the code snippet to access both parameters in the handler function.

Listing 3.5: Accessing multiple parameters

```
server.route({
  method: "GET",
  path: "/{userName}/lists/{listName}",
```

```
4    handler: (request: hapi.Request, reply: hapi.IReply) => {
5      console.log("User Name: " + request.params.userName);
6      console.log("List Name: " + request.params.listName);
7      reply("success");
8      }
9  });
```

3.4 Optional parameters

Hapi supports optional parameters in the path. We can use optional parameters to define a single route to fetch both a single resource and its collection. Optional parameters are indicated by appending a `?` to its name.

Listing 3.6: Optional parameters

```
1  server.route({
2    method: "GET",
3    path: "/users/{userId?}",
4    handler: (request: hapi.Request, reply: hapi.IReply) => {
5      if (request.params.userId) {
6        return reply("user id is: " + request.params.userId)
7      } else {
8          return reply ("will show user collection");
9        }
10     }
11 });
```

Optional parameter can appear only as a last parameter.

3.5 Wildcard parameters

If we want to repeat a segment, we can use `*` followed by a number. Say, we are running a blog and want to repeat the `category` twice. Then the path will be,`/{category*2}`.

We can `split` the segment to get the individual segments. For example, `request.params.category.split('/')` will get us the individual segment.

We can also omit the number to match unlimited segments. This is called `catch-all route` since they handle routes that are not handled by specific routes. They are useful in implementing `404` pages.

3.6 Route handlers

Handler function accepts two parameters: `request` and `reply`.

The `request` parameter contains headers, authentication information, payloads and others. The `reply` parameter is used to respond to the requests. Usually it only contains payload. But it can also have headers, content types, content length and so on. It can be chained to indicate the response code too. For example, `reply('not found').code(404)`.

3.7 Query strings

Query strings are another mechanism to send information from client to server. We saw that http://stackoverflow.com/questions/tagged/hapijs lists all the questions tagged with hapijs. If you want to access the same via Stackoverflow API, then the url is `https://api.stackexchange.com/2.2/que`