

Introduction

TypeScript

Info

TypeScript is a super-set of JavaScript and all plain JS is valid TypeScript that the TS compiler will compile any existing JS application without the need for adding any type annotations.

TypeScript compiles TS code into a target JS format. The target output can be configured with a `tsconfig.json` file in the root folder of the project. Targets include ES2015/E6+, commonjs and more.

Basic Types

Primitives: **string, number, boolean**

Special: **any, unknown, void, null, undefined**

Note on the 'any' type:

When you don't specify a type, and TypeScript can't infer it from context, the compiler will typically default to any.

You usually want to avoid this, though, because any isn't type-checked. Use the compiler flag 'noImplicitAny' to flag any implicit any as an error.

Functions

Without TS:

```
function greet(name) {  
    console.log('Hello ' + name + '!');  
}  
  
greet('Name') // Hello Name!  
greet(42) // Hello 42!  
greet({ a: 'apple' }) // Hello [Object object]!
```

With TS:

```
function greet(name: string): void {  
    console.log('Hello ' + name + '!');  
}
```

```
greet('Name') // Hello Name!  
greet(42) // Argument of type 'number' is not assignable to parameter of type 'string'.  
greet({ a: 'apple' }) // Argument of type '{ a: string; }' is not assignable to parameter of type 'string'.
```

Objects

Without TS:

```
const user = {  
    name: 'Name',  
    id: '1234',  
};  
  
user.address = '123 Fake St.';  
user.email = 'e@mail.com';
```

With TS:

```
const user: {  
    name: string;  
    id: string;  
    address?: string;  
} = {  
    name: 'Name',  
    id: '1234',  
}  
  
user.address = '123 Fake St.';  
user.email = 'e@mail.com';
```

Interfaces

```
interface User {
  id: string;
  created: Date;
  contact: {
    firstName: string;
    lastName: string;
    middleName?: string;
    address: {
      street: string;
      city: string;
      zip: string;
      country: string;
    };
  };
};
```

Types

```
type UserResponse = User | null

type CallbackFunction =
  (user: UserResponse, error?: Error) => void;

function getUser(id: string, cb: CallbackFunction): void {
  const user = database.getUser(id);
  if (!user) {
    return cb(null, new Error('No user found'));
  }
  cb(user);
}
```

Classes Array, Date, URL, Promise, etc.

```
class Storage {

  private users: Array<User> = [];

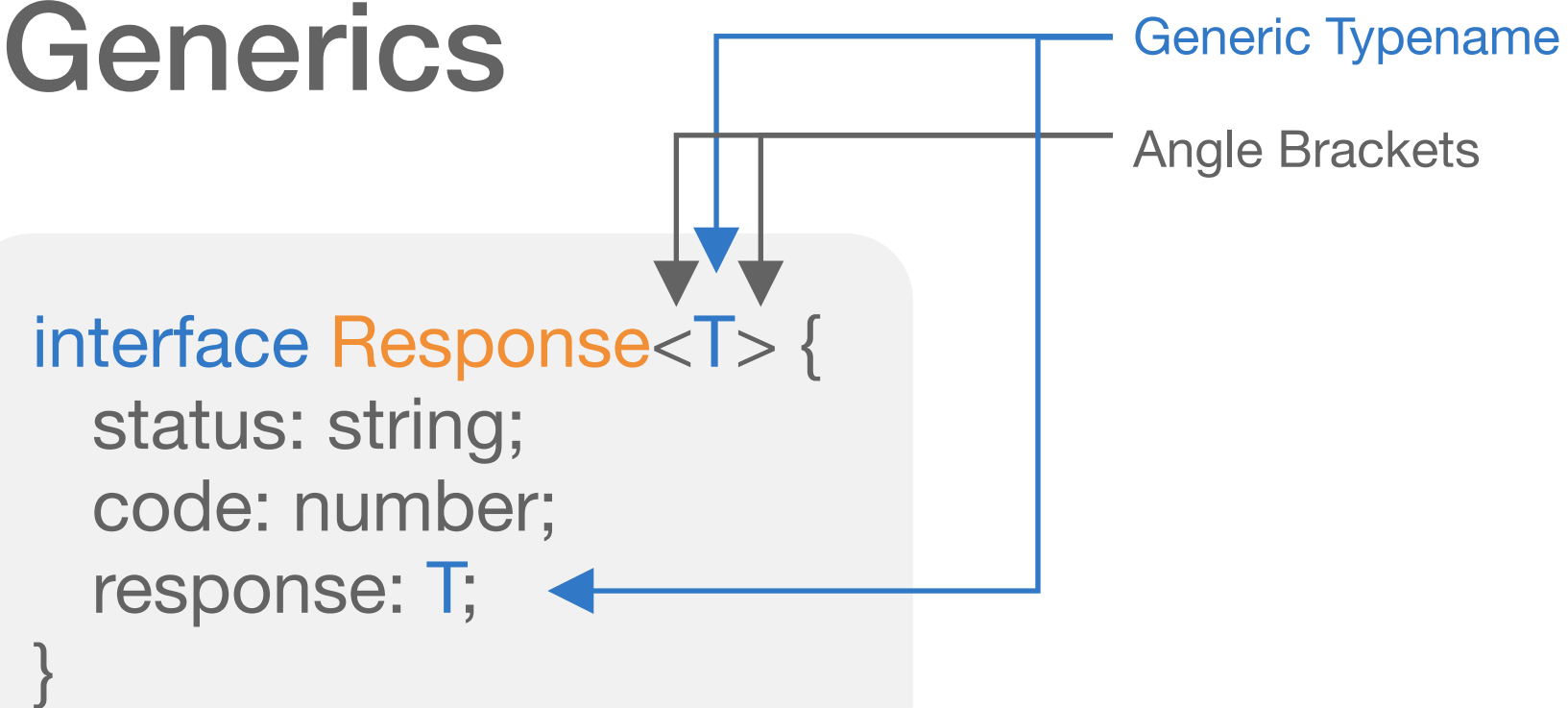
  constructor(private readonly name: string) {}

  static toString(s: Storage): string {
    let out = s.name + '\n';
    out += s.users.length + ' number of users';
    return out;
  }

  public addUser(u: User): void {
    this.users.push(u);
  }
}
```

Generic Types

Generics



```
type UserResponse = Response<User>;
```

```
function generateOkayResponse<T>(response: T): Response<T> {  
  return {  
    status: 'success',  
    code: 200,  
    response,  
  };  
}
```

```
function generateOkayResponseUser(user: User): UserResponse {  
  return {  
    status: 'success',  
    code: 200,  
    response: user,  
  };  
}
```

Generic Class

```
class Container<T> {  
  
  private items: Array<T> = [];  
  
  public push(item: T): void {  
    this.items.push(item);  
  }  
  
  public get(index: number): T | null {  
    const item = this.items[index];  
    if (item) {  
      return item;  
    }  
    return null;  
  }  
}
```

Array

You can annotate arrays as such:

```
const users: Array<User> = [];
```

```
const users: User[] = [];
```

Both are valid

Enum Types

```
enum Keys {  
  Up = 'up',  
  Down = 'down',  
  Left = 'left',  
  Right = 'right',  
  A = 'a',  
  B = 'b',  
  Start = 'start',  
  Pause = 'pause',  
  None = 'none',  
}
```

```
enum Actions {  
  Jump = 'jump',  
  Crouch = 'crouch',  
  MoveLeft = 'move:left',  
  MoveRight = 'move:right',  
  Punch = 'punch',  
  PickupItem = 'pickup:item',  
  None = 'none',  
}
```

Enums are one of the few features TypeScript has which is not a type-level extension of JavaScript.

Enums allow a developer to define a set of named constants. Using enums can make it easier to document intent, or create a set of distinct cases. TypeScript provides both numeric and string-based enums.

```
class Player {  
  
  private inventory: Array<InventoryItem> = [];  
  
  public onKey(keyString: string): void {  
    const key = this.getKey(keyString);  
    const action = this.getAction(key);  
    this.onAction(action);  
  }  
  
  private getKey(key: string): Keys {}  
  
  private getAction(key: Keys): Action {}  
  
  private onAction(action: Action): void {}  
}
```


Utility Types, Unions and Intersections

Builtin Utility Types

Record<Keys, Type>

Constructs an object type whose property keys are **Keys** and whose property values are **Type**. This utility can be used to map the properties of a type to another type.

```
type TemperatureMap = Record<string, Array<number>>;
const temps: TemperatureMap = {
  Denver: [5, 3, -2, 17],
  Tulsa: [63, 51, 43, 71],
  Chicago: [-5, -12, 2, 7],
}
```

Partial<Type>

Constructs a type with all properties of **Type** set to optional. This utility will return a type that represents all subsets of a given type.

```
interface Book { title: string; author: string; dates: Array<Date>; }
function updateBook(book: Book, update: Partial<Book>): Book {
  return { ...book, ...update };
}
```

Pick<Type, Keys>

Constructs a type by picking the set of properties Keys (string literal or union of string literals) from Type.

```
function bookPreview(book: Book): Pick<Book, 'author' | 'title'> {
  return { author: book.author, title: book.title };
}
```

Omit<Type, Keys>

Constructs a type by picking all properties from Type and then removing Keys (string literal or union of string literals).

```
function bookPreview(book: Book): Omit<Book, 'dates'> {
  return { author: book.author, title: book.title };
}
```

Unions and Intersections

Unions

```
interface Book { title: string; author: string; id: number; }
interface Album { title: string; artist: string; id: number; }
interface Painting { title: string; painter: string; id: number; }
type HasId = Book | Album | Painting; // { id: number }
```

```
function getItemById(item: HasId): Book | Album | Painting {
  return database.getItem(item.id);
}
```

Intersections

```
interface Book { title: string; author: string; }
interface Album { title: string; artist: string; }
interface Painting { title: string; painter: string; }
interface HasId { id: number; }
type BookWithId = Book & HasId;
// { title: string; author: string; id: number }
type Indexable<T> = T & HasId;
```

```
function index<T>(item: T): Indexable<T> {
  return { ...item, id: generateId(), };
}
```

TCP and the net module

NodeJS | TypeScript

TCP, Sockets and HTTP

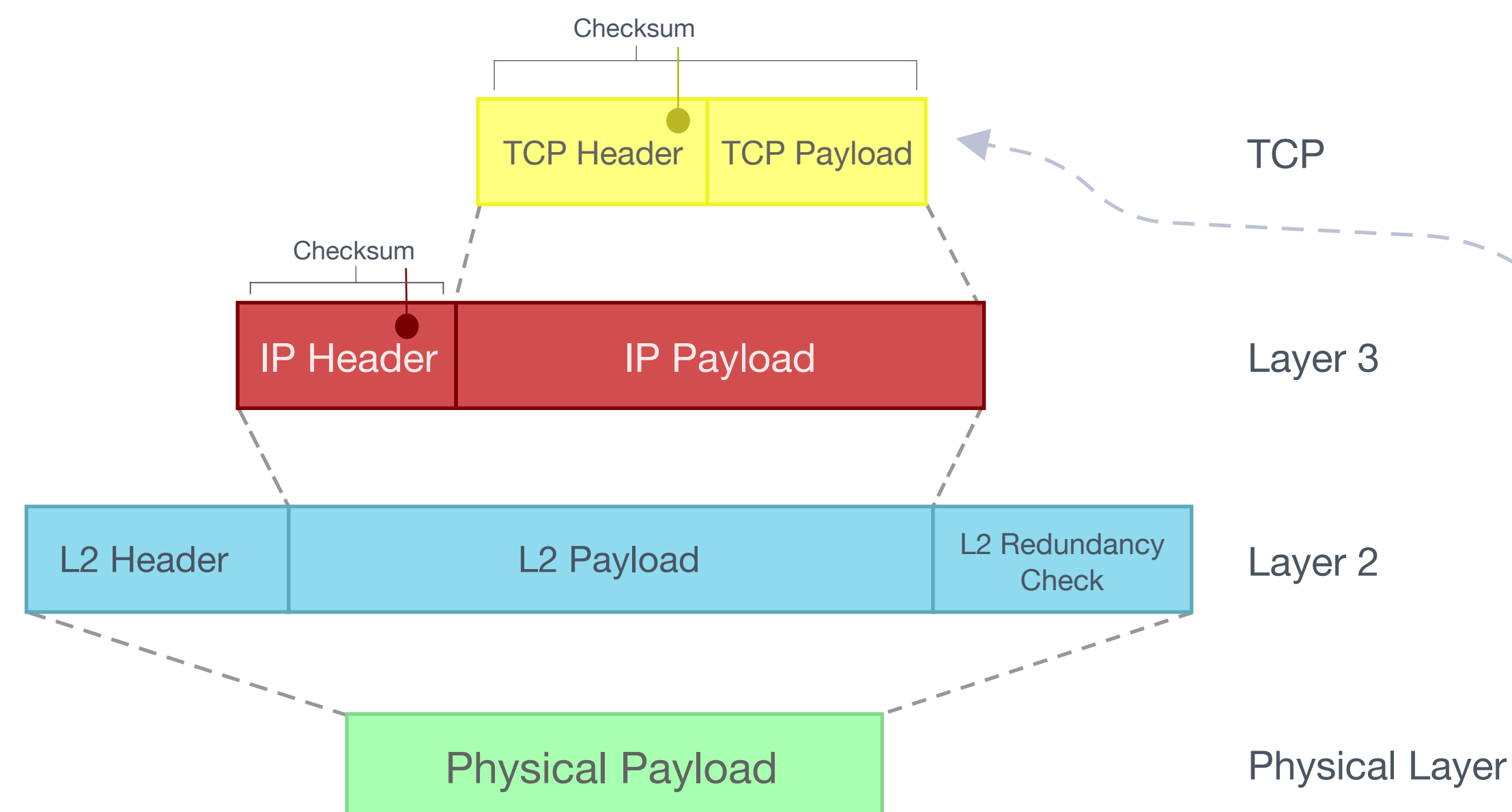
TCP

Transmission Control Protocol

TCP allows computers to exchange data over a network.

It does this by organizing the data in a way that allows for data integrity through error control coding, redundancy checks and a series of checksums. TCP will also resend packets.

TCP is used to build higher-level data-transmitting protocols like SSH and FTP and is also used to send emails through IMAP, POP3 and SMTP. HTTP also utilizes TCP for transmitting data across a network.



TCP Sockets

TCP Sockets are interfaces that allow programmers to send and receive data via TCP connections. This communication can be across a network or on your local machine, like connecting to localhost.

TCP connections need addresses to label and locate each other. These addresses are known as IP addresses. Example: 127.0.0.1 for IPv4 or FF01::101 for IPv6. TCP connections also require a port to connect to which allows the system to differentiate between various open sockets.

HTTP

HTTP is a client/server protocol for fetching resources across a network. Resources can be files, or parts of files such as html documents, images, sound files and more.

HTTP is built on top of TCP and uses it to send messages adhering to the HTTP standards. This consists of a status line, followed by headers and any query parameters, a newling and the body at the end.

```
HTTP/1.1 200 OK
Content-Type: text/html
x-application-key: 1234...

<html>
  <body><h1>Hello World</h1></body>
</html>
```

URI, URL & URN

URI

Uniform Resource Identifier

The target of an HTTP request is called a "resource", whose nature isn't defined further; it can be a document, a photo, or anything else. Each resource is identified by a Uniform Resource Identifier (URI) used throughout HTTP for identifying resources.

URL

Uniform Resource Locator

The most common form of URI is the Uniform Resource Locator (URL), which is known as the web address.

<http://example.com>

<http://example.com/users/1234/dashboard>

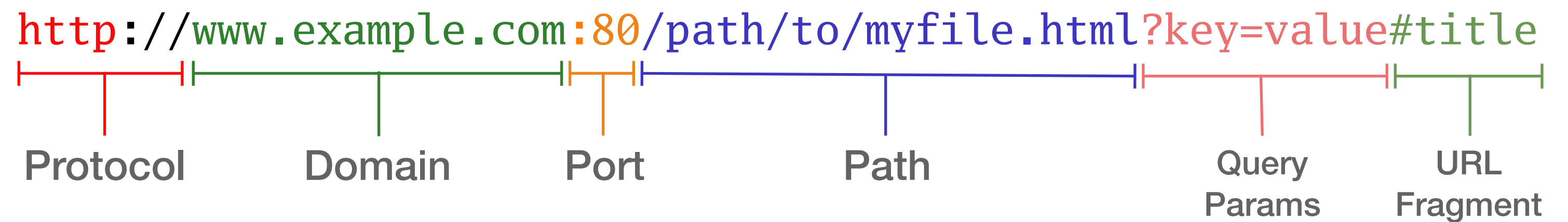
<https://example.com/users/search?size=10&q=2->

URN

Uniform Resource Name

A Uniform Resource Name (URN) is a URI that identifies a resource by name in a particular namespace.

<urn:isbn:9780141036144> - Nineteen Eighty-Four by George Orwell



Protocol

The protocol or scheme indicates how the incoming data should be parsed. Common schemes include http, https, ftp, data, tel, mailto, etc.

Domain

The domain or authority indicates the name of the server being requested. It is also possible to use an IP address as a domain name is just an alias that points to its server's IP address.

Port

Servers listen on ports denoted by a number, e.g. 3000, which allows multiple TCP servers on a single host. The port can be omitted from the URL if it is 80, this is the default port for HTTP servers.

Path

The path tells the server exactly where to look for the requested resource. This could represent a physical path to documents or an abstract path handled by the server.

Query Parameters

The query parameters are extra key/value parameters provided to the web server. The key/value pairs in the format `key=value` with each pair separated by a **&** sign.

URL Fragment

URL Fragments point to locations inside of the requested resource, like bookmarks. This part of the URL is not sent to the server and is used by the browser. Example usages are scrolling to an anchor tag inside an HTML document and skipping to a point in time in a video or audio file

NodeJS net Module

The net module provides an asynchronous network API for creating stream-based TCP or IPC servers.

This is a built-in module that comes with all versions of NodeJS and has methods for creating a server and connecting to a TCP server.

Socket Interface

The net.Socket class is the interface in which we interact with the TCP socket. It has methods to write to and read from the TCP stream, just like a file. It also gives the programmer the ability to pause and resume the data flow, get information about the connection and uses the JavaScript event system.

Creating a Server

```
import net from 'net';

const server = net.createServer((socket: net.Socket): void => {
  socket.on('data', (data: Buffer) => {
    const response: string = handleRequest(data);
    socket.end(response);
  });
});

server.listen(3000);
```

Connecting to a Server

```
const client = net.createConnection({ port: 3000 }, () => {
  client.write('I connected!');
  client.end();
});

client.on('data', (data: Buffer) => {
  console.log(data.toString());
});
```

Basic HTTP server using net


```
import net from 'net';

const response = `HTTP/1.1 200 OK
Content-Type: text/html

<html><body><h1>Hello World</h1></body></html>`;

const server = net.createServer((socket: net.Socket): void => {
  socket.on('data', (data: Buffer) => {
    socket.end(response);
  });
});

server.listen(3000);
```



```
$ curl http://localhost:3000
<html><body><h1>Hello World</h1></body></html>
```

BongoDB

TypeScript

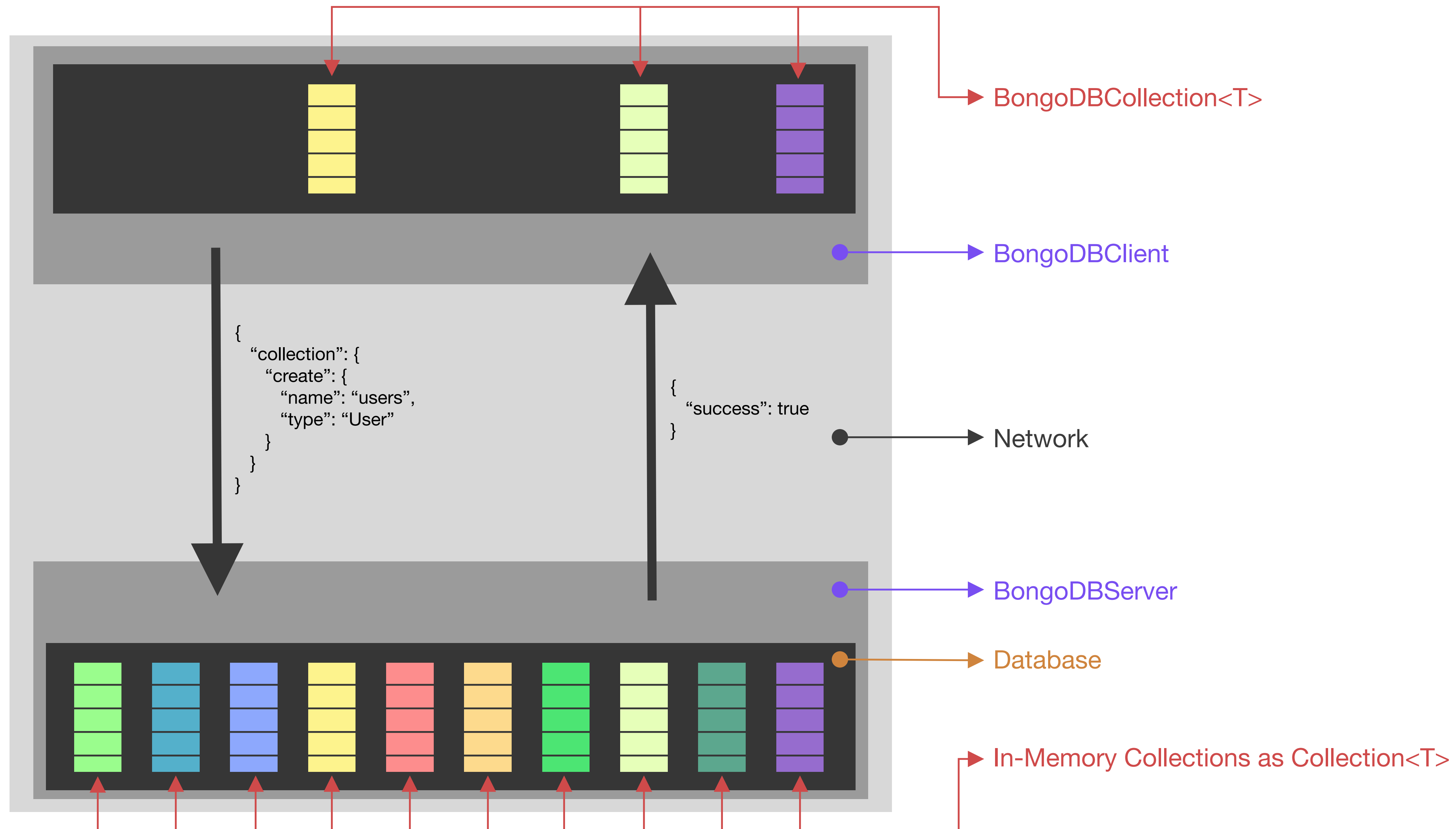
In putting this knowledge to practice we will build a simple in-memory database called **BongoDB** that we can connect to over a network to run CRUD commands.

This database needs to be generic and must have a client that can connect to a server to create/read stored entries.

The **Database** will manage **Collections** which operate on **Entries**.

The first step is planning out how this database will work and decide what parts we need to build and when. The second step is setting up a NodeJS project with Typescript and then following through with the plan.

Project Map



Project 1 - Database/Collection

Collection

```
class Collection<T> {  
    private items: Array<T> = [];  
  
    constructor(public name: string, public type: string) {}  
  
    create(item: T): T {}  
  
    read(query: Partial<T>): T[] {}  
  
    update(query: Partial<T>, updates: Partial<T>): T[] {}  
  
    delete(query: Partial<T>): T {}  
  
}
```

A collection manages entries (of type T). The collection keeps the list of entries private and you must interact with the entries via the methods on the collection.

Database

```
class Database {  
    private collections: Record<string, Collection<unknown>> = {};  
  
    public createCollection<T>(name: string, type: string): Collection<T> {}  
  
    public readCollection<T>(name: string): Collection<T> {}  
  
    public updateCollection<T>(name: string, newName: string): Collection<T> {}  
  
    public deleteCollection<T>(name: string): void;  
  
}
```

A database holds an object with every collection it under its control. Each collection can be accessed using its name as the key. The database keeps the collections private and exposes access via the CRUD methods.

CRUD
Actions