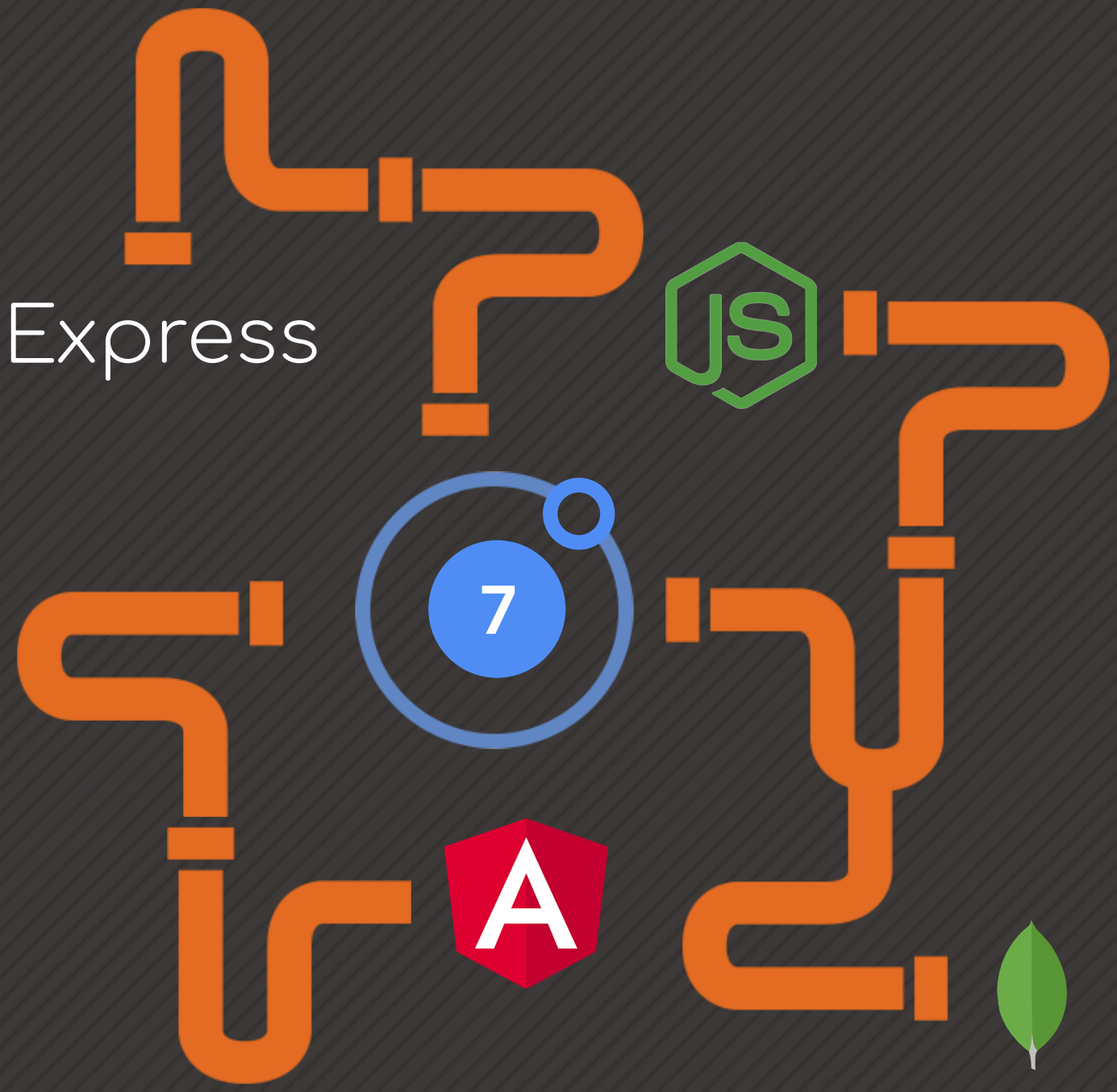
Mastering Ionic

Working with the MEAN stack



James Griffiths

Mastering Ionic - Working with the MEAN stack

Saints at Play Limited

Thanks to...

The teams at Ionic and Angular for creating such phenomenal products that allow millions of developers worldwide to realise their ideas quickly and easily.

The awesome developers and communities behind PHP, MySQL, SQLite, Firebase, PouchDB, CouchDB, Docker, NodeJS, ExpressJS, MongoDB, Mongoose, ElectronJS and rxJS.

The developers, contributors and drivers behind all of the JavaScript/TypeScript packages and libraries used within the projects covered in this ebook - you guys are awesome!

Every developer who ever helped answer a question that I had or a software bug that I was trying to fix - I may have forgotten many of your names but I will always appreciate the assistance you have provided.

Those who believed in me and gave me a chance to shine when many others didn't or just wouldn't - you are not forgotten!

God above all others - without whom nothing would be possible.

Table of Contents

Introduction

Thank you for purchasing this digital copy of Mastering Ionic: Working with the MEAN stack.

My goal with this ebook is to guide you through working with the MEAN stack (MongoDB database, Express middleware routing, Angular and NodeJS) to seamlessly integrate data into an Ionic application.

We start with exploring database concepts and terminology - for both SQL and NoSQL databases before progressing onto developing/publishing a MEAN powered Ionic/electron desktop application.

We'll cover working with the MongoDB database and Mongoose API to provide CRUD (Create, Read, Update and Delete) functionality for our Ionic applications.

Along the way we'll work with a range of technologies and services including:

- CapacitorJS
- MongoDB/Mongoose
- NodeJS
- Express Middleware routing
- rxJS library methods

We'll cover tips for best practice, discuss known limitations and/or potential challenges with the services/tools that we are using and I'll provide you with further resources to reinforce what you've learnt within the pages of this e-book.

No matter what your level of experience working with the technologies that we'll be covering, I hope you find this e-book both useful and enjoyable and I look forward to receiving your feedback.

What this book isn't

If you're looking for an in-depth technical guide into all the functionality and features available within MongoDB, Express and NodeJS then this simply is NOT the book for you.

What is covered

This book provides detailed information on the core essentials of MongoDB, Express and NodeJS, creating API endpoints, performing CRUD related operations and how to integrate data from MongoDB into an Ionic frontend.

I promise you there will be plenty of useful real-world information that you can take and use within your own projects/applications but a deep-dive into the MEAN stack (and all of its core features) is simply not possible (as the book would have to be at least twice as large…and even then it would only be scratching the surface).

Prerequisites

I am assuming (hopefully not wrongly!) that you already have, at the very least, a basic understanding and level of familiarity with developing projects for web, iOS & Android using the Ionic CLI.

You will also need to be familiar with understanding and being able to use HTML5, Sass/CSS and Angular/TypeScript which are core languages/frameworks used within Ionic.

I won't be covering those languages/frameworks in depth (other than demonstrating how they can be used throughout the projects we'll be developing) so, if you do require any background information/further instruction on their usage, a good place to start would be with the following resources:

- Angular
- TypeScript
- HTML5
- Sass
- CSS

You will also need to have some familiarity/experience with command line usage as we'll be creating projects, page components & Angular services, installing the required plugins and software libraries with the Ionic CLI.

Last, but not least, you'll also need to have a basic understanding (as well as some

experience) with object oriented programming (otherwise referred to by its acronym of OOP) as TypeScript is a class based OOP language (actually a superset of JavaScript if you want to get technical).

If you're not all that familiar with Object Oriented Programming (in the context of TypeScript/JavaScript) then I would recommend starting with the following online resource which should help get you up to speed.

So who am I and why should you listen to me?
I'll answer the last part of that heading first....only if you feel you want to!

Joking aside my background in web/mobile development stretches back to 2002 when developing online projects, almost exclusively in the form of websites (as the iPhone was still 5 years away and mobile development was, to put it mildly, an extremely small niche due to limited possibilities with the available technologies, tools and device/browser support - anyone reading this remember WAP?), was in a relatively nascent stage.

Even though largely forgotten and, in many developer circles (for those of us who are old enough to remember) widely derided and scorned, Macromedia Flash MX was my introduction to developing websites and applications (albeit only browser and CD/DVD-ROM based at the time).

I'm probably going to invite ridicule and exasperation with the following statement but I loved working with that software and the possibilities it opened up for creative experimentation, programming and designing/developing applications.
As the first decade of the twenty-first century progressed, and browser support for language standards and features improved, it became more and more apparent that Flash had certain limitations that working with HTML/CSS (and the re-emergence of JavaScript as a scripting language - helped with frameworks like jQuery and MooTools as well as a trend towards consistent browser support) didn't.

This gradually led to more frontend focussed development as well as incorporating PHP/MySQL into my digital toolbox.

Fast forward to 2010 and I'm starting my initial journey into developing mobile applications with jQTouch and PhoneGap, subsequently followed by jQuery mobile before finally settling on Ionic in 2014.

Along the way I've delivered websites and mobile/tablet applications for a variety of clients including Halco Energy, West Midlands Police, Maplecroft, WQA, Virgin Media, EDF Energy, Evans Cycles, Shelter and the British Science Association as well as various digital agencies, marketing companies and small business clients.

I'd like to think, as a result of the past 20+ years, that I've accumulated a certain wealth of experience, knowledge and skills that can be shared with the wider development community in the form of my blog and, of course, my e-books.

I certainly don't consider myself to have reached any vast summit of knowledge and, in some respects, I feel like I'm only just starting to scratch the surface of discovering what's possible with all these incredible web based technologies that we have access to now.

Just like yourself I'm still on a journey of learning, growing and maturing as a developer...and with the rate of technological change that is continually taking place there's always more to learn!

Support

So you've purchased my e-book (or are maybe considering making a purchase) and you might find yourself with some questions regarding how often the content is kept updated with changes to Ionic and/or the associated technologies that are covered in these pages, what help/assistance is available with possible development issues you might encounter and what further resources you might be to access.

Firstly, schedule permitting (although even the best will in the world can be thwarted by external events and circumstances), I endeavour to keep the e-book content updated within 7 days of significant changes to Ionic and/or featured databases and technologies.

I routinely e-mail my customers with news of updated e-book content that has been

published.

Finally, all downloadable code examples for each chapter and the featured case studies are available here.

Conventions used within this e-book

Fortunately there's only a small number of conventions employed within this e-book that you need to be aware of.

ALL of the code examples that are featured in each of the chapters and case studies are displayed within a grey rectangle, which may (or may not) contains additional comments (rendered in italics), like so:

```
// Install the required platforms
npx cap add electron
```

Important information that requires your full attention is prefixed in its own paragraph like so:

IMPORTANT

Previous code examples that have been covered/explained will, where further additions to that script are required, be rendered with a placeholder in the following format:

...

There will, due to the limitations imposed by the width of the page dimensions of this e-book, be instances where code might run onto other lines. Where this occurs a hyphen will be inserted into that line of code to indicate that the displayed code is all part of the same line.

Use of hyphens in this specific context do NOT form part of the code logic but merely demonstrate that the code is continuing from one line to the next.

Where external resources are mentioned/used within each chapter these are

rendered in the form of hyperlinks along with an additional list of those hyperlinked resources displayed at the end of each chapter.

Finally, each chapter will, where necessary, conclude with a summary of the key concepts and information that has been covered.

IMPORTANT: You'll notice, as you go through the code examples covered in each chapter and case study, that I employ the following practice:

- Use of JSDoc syntax for commenting project component and service classes
- Specific naming conventions for class properties and methods (to help readily identify, or hint at, the purpose of that segment of code) • Formatting the code so it is more easily readable

You don't have to adopt the same practice (as each developer will have their own specific coding/formatting style) but it is a good idea to invest time into making your code as understandable/readable as possible (which is why I employ the above approaches in this e-book as well as my own digital projects).

After all, if you come back to a project 3 or more months later (or are working with other developers), such efforts will help make managing the project quicker and easier in the long run - and that can't be a bad thing (especially if you happen to forget why you coded something in a certain way!)

Glossary

Technical terms

Technical glossary

To wrap up the introduction to this e-book let's quickly cover some of the keywords and terms that we'll be encountering/using over the following chapters.

I imagine most of you will already be familiar with these so feel free to press on to the next chapter if that's the case! If not, please take a few minutes to read through the following terms and familiarise yourself with their meaning.

ACID

Acronym for Atomicity, Consistency, Isolation & Durability - a measure used to determine how effective a database system is as at performing transactions

Angular

A front-end component-based framework for building scalable web applications that is the default choice of framework for Ionic

API

Application Programming Interface - A set of tools for a particular software library, framework or service that developers can utilise in their own projects

Authentication

The act of verifying that a supplied identity is genuine

Authorisation

The act of granting access to a system or service

Backend as a Service

Often referenced as the acronym BaaS refers to a cloud computing model which allows web/mobile application developers to connect with services such as cloud storage, push notifications and NoSQL databases through the use of dedicated API's/SDK's

BASE

Acronym short for **B**asically **A**vailable, **S**oft state, **E**ventual consistency - a data consistency model used by many NoSQL databases

CapacitorJS
A cross-platform runtime API similar to Apache Cordova that is focussed on performant mobile applications that adhere to Web Standards while accessing native device functionality on platforms where support is available

Content Delivery Network
A Content Delivery Network, often abbreviated as CDN, is a system of distributed servers, spanning multiple geographical locations, that allows websites and applications to benefit from high availability of content, low network latency and improved performance

Class based programming
A style of Object-Oriented Programming where objects are generated through the use of classes

CLI
Command Line Interface - A software utility that allows commands to be executed solely through text input

CRUD
Acronym for Create, Read, Update and Delete, which are common operations performed on data

Electron
An application development framework that allows users to build cross-platform desktop applications using HTML, CSS & JavaScript

Firebase
A Google owned/managed BaaS platform which provides a variety of cloud related services such as Authentication, Storage, NoSQL databases & Push notifications

Hybrid Apps
Mobile applications that are typically developed using web based languages such as HTML, CSS and JavaScript which are then able to be published within native mobile wrappers for deployment to iOS, Android, Windows Mobile devices etc

Ionic Framework
An open source application development framework for developing progressive web apps and mobile applications

JSON
JavaScript Object Notation - A subset of the JavaScript programming language that specifies/provides a standard for exchanging data

Node
An open-source, cross platform JavaScript environment for developing server-side web applications

NoSQL
A type of database where data is typically stored in the form of JSON objects

Object Oriented Programming
Often referenced by its acronym of OOP - A type of programming where code is developed based around the concept of objects and their relationship to one another

Package Manager
A tool, or collection of tools, for managing the installation, configuration, upgrading, removal and, in some cases, browsing of software modules on a user's computer

SDK
Software Development Kit - A suite of development tools that developers can use with a particular software program, library or platform

Transaction
A unit of work performed within a database system

Databases

A short summary

In its simplest definition a database is a structured container for storing data and allowing that to be acted upon (I.e. CRUD related operations, importing/exporting data and performing searches).

Databases come in a variety of models (and often implement schemas).

Models and schemas

A database model determines the logical structure of a database including the relationships and constraints of how data is able to be stored and accessed.

Common database models include:

- Hierarchical
- Relational (aka Relational Database Management System [RDBMS] or SQL database)
- Non-relational (NoSQL)
- Object-oriented
- Network

Of these the Relational and Non-relational (NoSQL) database models are the most commonly used - at least as far as most organisations/developers are concerned.

A database schema refers to the logical grouping, organisation and structure of objects that are used within the database (such as tables, views, indexes, stored procedures etc).

For example, a database model may define the overall structure of the database and how data is stored/accessed (i.e. relational or NoSQL) yet there may be one or more schemas defining the structure, organisation and relationships between certain parts of that database system (i.e. accounting schemas, auditing schemas, reporting schemas etc)

Models and schemas can often be confusing to define as they are sometimes used interchangeably or given slightly different meanings with some database systems.

Relational databases

A relational database model structures, groups and organises data using:

- **Tables** (structures that impose a schema on the records that they contain)
- **Rows** (the individual records that are stored within a table)
- **Columns** (the distinct fields that data is stored under for each record)

Fields come in many different data types (with potential constraints and additional flags depending on the data type being supported) which may include - for example:

- integer
- float
- text
- date
- boolean

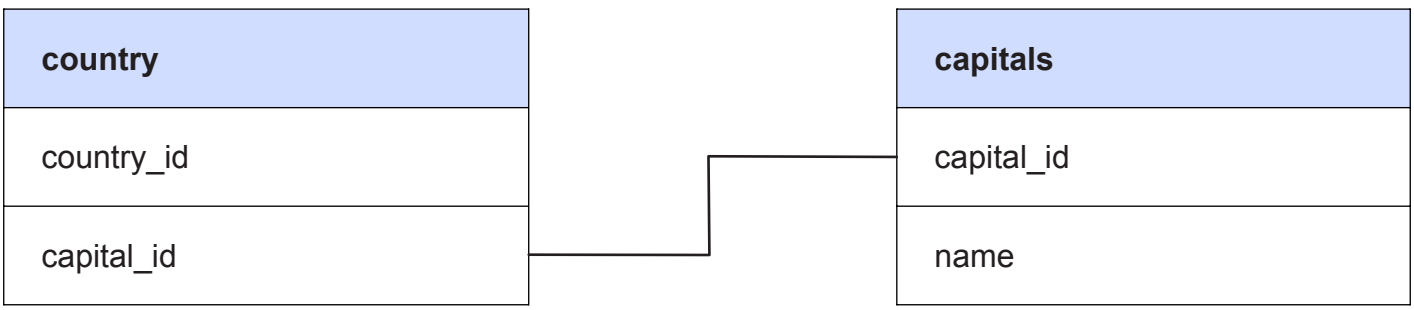
Relationships

Tables can establish relationships with one another through the use of keys:

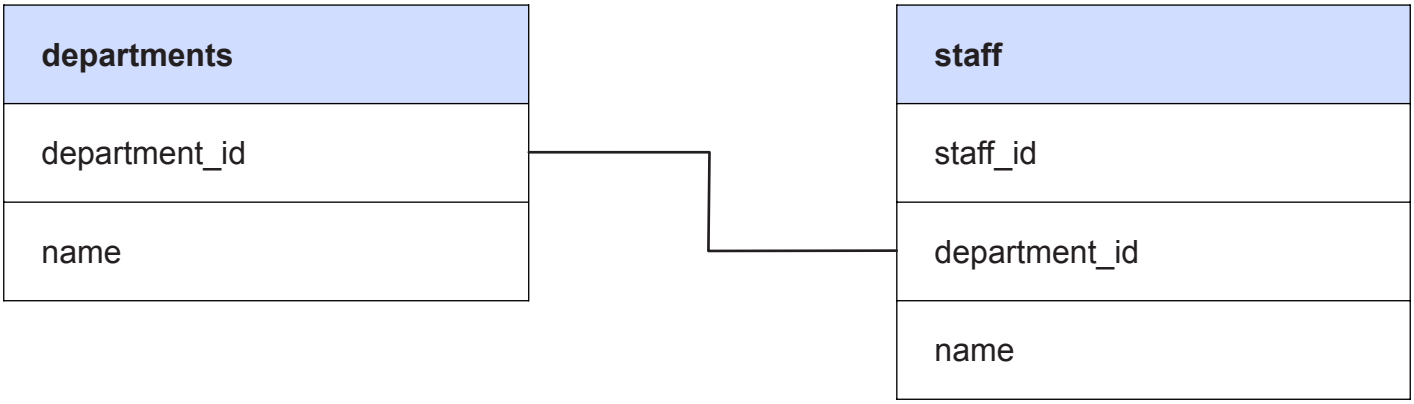
- **Primary key** - A table column where each record has a unique value
- **Foreign key** - A table column whose values references the primary key of another table

Within an RDBMS there are 3 possible types of relationships between tables.

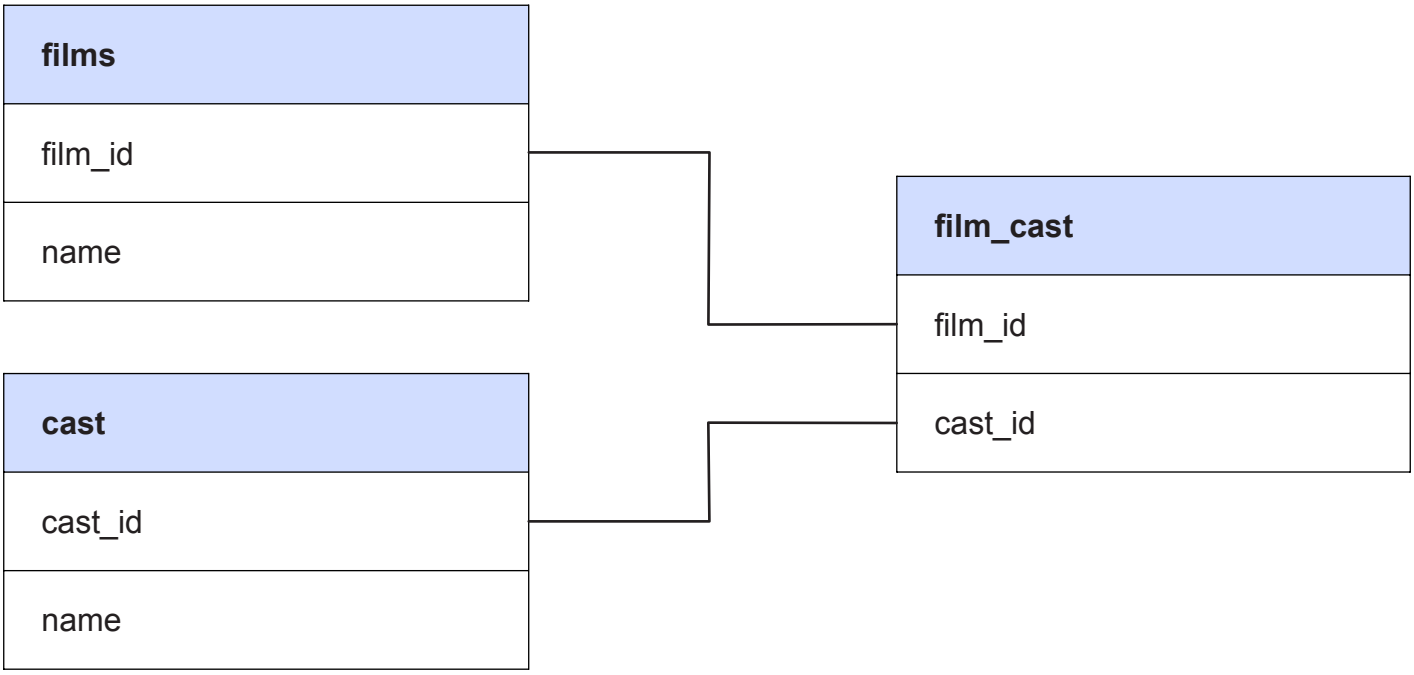
A **One-to-one** relationship consists of a single record on each side - for example, the relationship between a country and its capital city (as a country will only ever have one capital):



A **One-to-many** relationship consist of a single record on one side and many records on the other side. This could occur, for example, in a company where an employee belongs to a single department but a department can have many employees:



A **Many-to-many** relationship consists of multiple entries on both sides of the relationship. This could be represented in a movies database for example where a single movie can feature many cast members and a cast member can feature in many movies:



Normalisation

When designing the data architecture for a SQL database system (I.e. determining the quantity, types and nomenclature of fields used within the different tables and the purpose of each table) it's important to minimise data duplication to make data entry more efficient and easier.

For example if there are two or more tables that contain a product name column it would make more sense to link these together so that data only needs to be entered once and not multiple times for the same item.

This process of reducing data duplication to make a maximally efficient relational database system is known as **normalisation**.

The use of **normal forms** - a guiding set of stages for which a database achieves ever greater levels of normalisation - was introduced by computer scientist Edgar Frank Codd in the early 1970's.

These normalisation guidelines consist of six normal forms although, typically, if a database is in **Third Normal Form** (3NF) it is generally considered to be normalised.

You can read more about database normalisation here.

SQL Queries

Users can query data within a relational database management system using SQL (Structured Query Language) which can be simple in its instruction or increasingly complex depending on the requirements and granularity of the query (which is where normalisation helps by making the system more performant).

A simple query such as retrieving all book records from a single books table in descending order of entry (i.e. most recent first) might be written as follows:

```
SELECT * FROM books ORDER BY book_id DESC
```

The SQL syntax is relatively simple and intuitive so that even to a non-database specialist the purpose of the query would be relatively self-explanatory.

A more complex SQL query - say retrieving a list of all films, their cast members and the studio responsible for producing that film where ticket sales grossed over $25 million dollars might look something like the following:

```
SELECT movies.id, movies.title, cast.id, cast.name, studio.id, studio.name
FROM movies INNER JOIN cast ON cast.movieId = movies.id INNER JOIN
studio ON studio.id = movies.studioId WHERE movies.id IN (SELECT movieId
FROM movies GROUP BY sales HAVING COUNT(*) > 25000000)
```

As you can see even with more complex queries SQL is relatively intuitive and easy to grasp (although some of the more complex query logic can take time to mentally parse - especially when working with different types of JOIN statements to draw data from multiple tables).

To learn more about SQL visit this online resource.

ACID

Databases that perform transactions in a timely, reliable and efficient manner are said to be ACID complaint.

ACID is an acronym for:

- **Atomicity** (all parts of a database transaction work as expected)
- **Consistency** (database transactions are performed as expected with no deviation in behaviour)
- **Isolation** (multiple transactions can be performed concurrently without affecting one another)
- **Durability** (data is saved with successful transactions even if a system failure or power outage occurs)

Relational database systems are widely trusted due to their ACID compliance.

Real world usage

Unsurprisingly many developers and organisations make use of relational databases to deliver their products/services (and you may likely have worked with such databases in an educational and/or professional context).

Such widely used SQL databases include (but are not limited to):

- MySQL
- PostgreSQL
- MariaDB
- SQLite
- Oracle
- Microsoft SQL Server

If you've ever developed applications using PHP then MySQL (and possibly PostgreSQL and MariaDB - a fork of MySQL) will likely be somewhat familiar to you.

To summarise then:

Relational database overview	
Pros	**Cons**
ACID (Atomicity, Consistency, Isolation & Durability) compliant - ensures that a database transaction is completed accurately and in a timely fashion	More time-consuming and difficult to modify pre-existing database architecture due to constraints in the database model
Support for table joins	High volume transactions can result in decreases in performance
Ability to perform complex queries	Difficult to scale with large amounts of data
Rigid, predictable structure for data	Does not work well with unstructured data
Allows for many different data types	Data normalisation can result in performance penalties

That concludes our brief and very basic introduction/overview of relational databases (and there's a lot more to learn…but for this book I want to keep things relatively simple and focus only on what we need to know for working with Ionic) so we'll now perform a similar walkthrough with NoSQL databases.

Non-relational databases

Ironically one of the strengths of relational databases is also experienced by many organisations and developers as one of its significant weaknesses: a rigid database architecture.

This rigid structure can be time-consuming, expensive and difficult to subsequently modify should even minor changes in data architecture be required (such as, for example, the addition of further fields or a change in the data types for existing fields).

Given that modern applications consume vast amounts of data (often supplied through third-party APIs) in the form of JSON objects SQL databases are not best suited for storage of, nor scaling with, such data formats.

Non-relational, more popularly known as NoSQL, databases were developed to address and meet these particular needs (amongst others).

Types of NoSQL

Instead of using tables non-relational databases store data using different models:

- **Column-based** (data is stored by column not row)
- **Document** (data is stored, often similar to JSON objects, in documents)
- **Graph** (data sets are represented as nodes, edges and properties with relationships represented as edges - or lines - between nodes)
- **Key-Value** (items of data are stored as key-value pairs within the database)

Data is typically stored as JSON, BSON (Binary JSON) or XML depending on the type of NoSQL database and the schema(s) that are supported.

In subsequent projects we will be working with document oriented NoSQL databases and will explore their data storage model that uses collections, documents and fields.

As there are a variety of NoSQL database models there is no one uniform language (unlike SQL with relational database systems) that can be used to perform queries across different systems.

For example Neo4J uses its own SQL inspired language called Cypher Query Language which is designed to work with its graphing NoSQL database model.

Using Cypher Query Language we could (in a hypothetical Neo4J database), for example, run the following query to retrieve all movies starring Keanu Reeves:

```
MATCH (keanu:Person {name: 'Keanu Reeves'})-[r:ACTED_IN]->(movie:Movie)
RETURN keanu, r, movie
```

Looks somewhat similar to SQL doesn't it?

If we are using NoSQL document-oriented database MongoDB however we would perform queries (in this case adding a new document to an hypothetical **users** database collection) using Mongo Query language (MQL) like so:

```
db.users.insert({
  id: "aebd4fs001",
  surname: "Bloggs",
  first_name: "Joe",
  email: "joe.bloggs@joebloggs.com",
  age: 25,
  status: "Active"
})
```

Notice how even though this is called a query language it looks nothing like traditional SQL or Cypher but shares the same dot syntax approach as working with JavaScript?

Unlike relational databases it is more difficult to port data between different NoSQL databases due to the different models that are used (document, graph, key-value and column).

Dropping ACID?

Many NoSQL databases sacrifice **atomicity** (where the integrity of the entire database transaction is guaranteed - not just a certain part of the transaction) or **consistency** (where database transactions are only successful where they meet certain database rules) in order to achieve high performance/scalability.

Although these features make SQL databases highly reliable they can present performance problems with high data/traffic usage as well as scaling issues - both of which NoSQL databases are adept in addressing and overcoming.

Typically, in lieu of ACID compliance, many NoSQL databases offer BASE properties:

- **B**asically **A**vailable - In the event of failure the system is guaranteed to be available
- **S**oft state - Data state could change without user interaction due to eventual consistency
- **E**ventual consistency - Consistency is not guaranteed at the transaction level but, after application data input, the system will be eventually consistent once data has replicated to all database nodes

Although BASE offers less assurances than ACID it effectively handles rapid data changes and scales well.

Not all NoSQL databases are non-ACID compliant but this may be an issue with organisations who require ACID compliant database in their day-to-day operations.

Advantages of NoSQL

- Can accommodate flexible data structures
- Designed to efficiently handle larger volumes of data
- Designed with an architecture to allow scaling for greater traffic
- Can be faster to develop with as the data structure allows for changes in response to modern agile development practices (with sprints, iterations and more frequent code changes)

- NoSQL data tends to integrate more easily, due to its structure/syntax, with JavaScript in modern cross-platform applications
- Polyglot persistence - allows for use of multiple data storage models within an application (i.e. using document and graph databases for separate areas of an application that require different data models)
- Minimises impedance mismatch - a term used to describe the difference between the relational model of the database and the structure of the data that is being saved (for example graphing data being saved in a tabular format - as in relational database systems)

Disadvantages of NoSQL

- Difficulty in porting data between different types of NoSQL database
- Many NoSQL databases do not offer ACID compliance
- Schema-less architecture can be off-putting to some developers concerned with maintaining data integrity
- Data is denormalised which requires mass updating (I.e. when changing a product image)
- No standardised query language across different database models

Real world usage

Given the flexibility of architecting data structures, increased data scalability, integration with modern web applications using JavaScript and performance it's not surprising that NoSQL databases have grown in popularity in recent years.

Some of the more widely used NoSQL databases include (but are not limited to):

- MongoDB
- Cloud Firestore
- Redis
- DynamoDB
- Apache Cassandra
- Neo4J
- PouchDB

MongoDB and Cloud Firestore are largely familiar with many frontend/mobile app developers due to their strong integration with JavaScript based technology stacks (and these, as you may remember from the contents page, along with PouchDB will be the NoSQL databases that we'll be working with in the pages of this ebook).

To summarise then:

Non-relational database overview	
Pros	Cons
Handles structured, semi-structured and unstructured data efficiently	Schema-less architecture (data integrity enforced from application not database)
Scales well with massive data storage	Not always ACID compliant
Scales well with cloud computing architecture	Difficult to port data between different NoSQL database models
Allows for multiple different types of data structures to be implemented	No standardised query language
API/data structure integrates well with JavaScript in modern cross-platform apps	
Reduced query times due to data architecture	

This then concludes our brief and very basic introduction/overview of Non-relational (or NoSQL) databases.

Unfortunately we are simply not able to cover each NoSQL database model in this book (**graph**, **key-value**, **column** and **document** - due to the volume of concepts and resources that would need to be covered).

As a result of this I have deliberately focussed on document oriented solutions as these are the most widely used NoSQL database model for many developers.

Resources

There's a lot more to learn about databases where SQL and NoSQL are concerned, and we've only scratched the surface covering the basics with this brief chapter.

Further database resources can be explored here:

- Structured Query Language
- Types of databases
- NoSQL

MEAN

Building a YouTube playlist

The MEAN stack is a popular open-source development stack built on JavaScript that uses the following four key technologies :

- **MongoDB** - NoSQL object-oriented database
- **Express** - A web application framework providing routing and middleware functionality for more seamless front-end and backend integration
- **Angular** - Front-end development framework (that we're familiar with as Ionic developers)
- **NodeJS** - A backend runtime environment that allows JavaScript to be used outside of the browser for server side scripting and writing command line tools

One of the many benefits of using the MEAN stack is the use of a single unified language across all of the different layers of that stack: JavaScript (or, in the case of Angular, as the superset of JavaScript known as TypeScript).

This unified language approach helps with development as we don't need to switch between - and learn/know - different languages for each tool (i.e. such as PHP or SQL with the server-side LAMP stack). This also has the added benefit of not needing to familiarise ourselves with different configurations as each of the tools in the MEAN stack can be installed/set-up using the same, or very similar, tools (homebrew and npm for example).

In the ever-evolving world of digital technologies you might also see similar, albeit competing, JavaScript development stacks based on newer tools/frameworks (or different approaches with existing technologies) such as:

- MERN (Mongo, Express, React & Node)
- MEVN (Mongo, Express, VueJS & Node)
- JAM (JavaScript, API & Markup)

Each of these different technology stacks has its pros & cons and, as with all tools, a lot of this is determined by your working environment (if part of a development team) and personal preferences as a developer (for example, I love working with Angular but cannot stand React).

In this chapter we're going to be focusing on using the MEAN stack to develop an

Ionic application that leverages a Node/MongoDB backend using Express for managing API routes and middleware functionality with all HTTP requests.

What we'll be creating

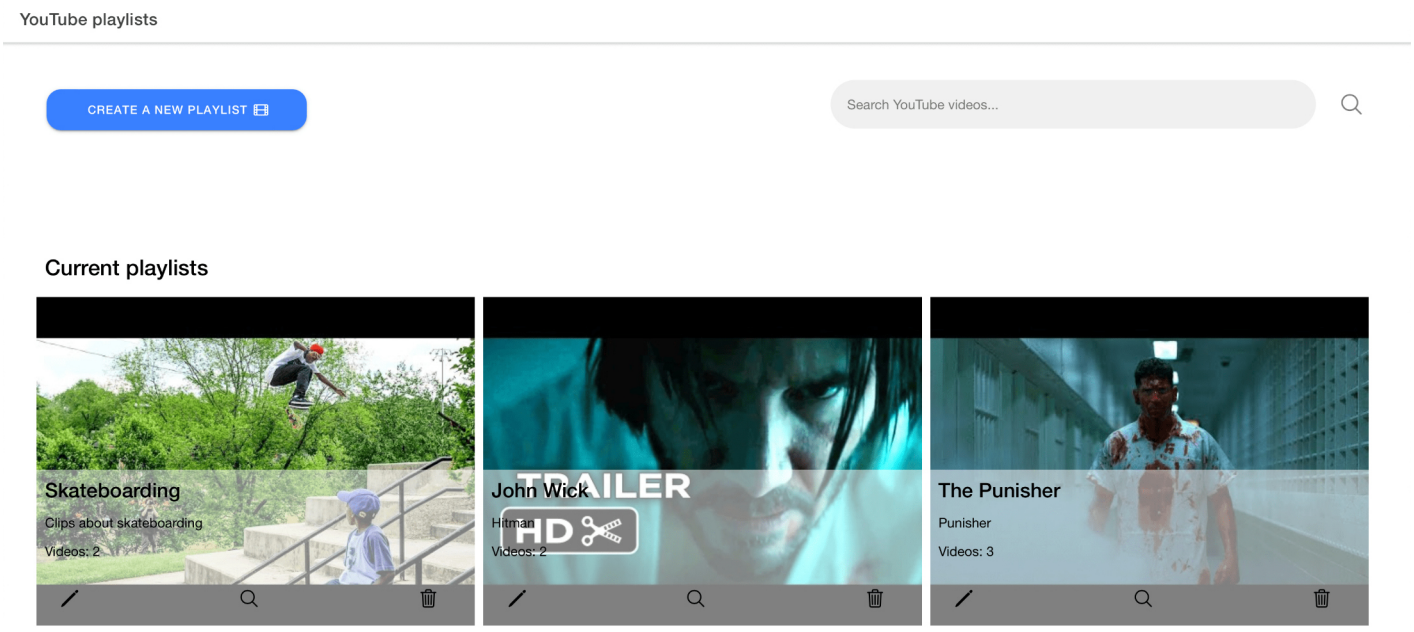
Our Ionic/MEAN app will consist of a single page application that offers users the following features and functionality:

- Perform YouTube searches
- Display/create/amend/delete video playlists
- Ability to assign returned YouTube videos to a selected playlist
- Remove videos from playlists
- Save playlists automatically to MongoDB

We'll develop our database driven application, in addition to using the MEAN stack, with the following third-party services/tools:

- YouTube API
- NGX pagination plugin

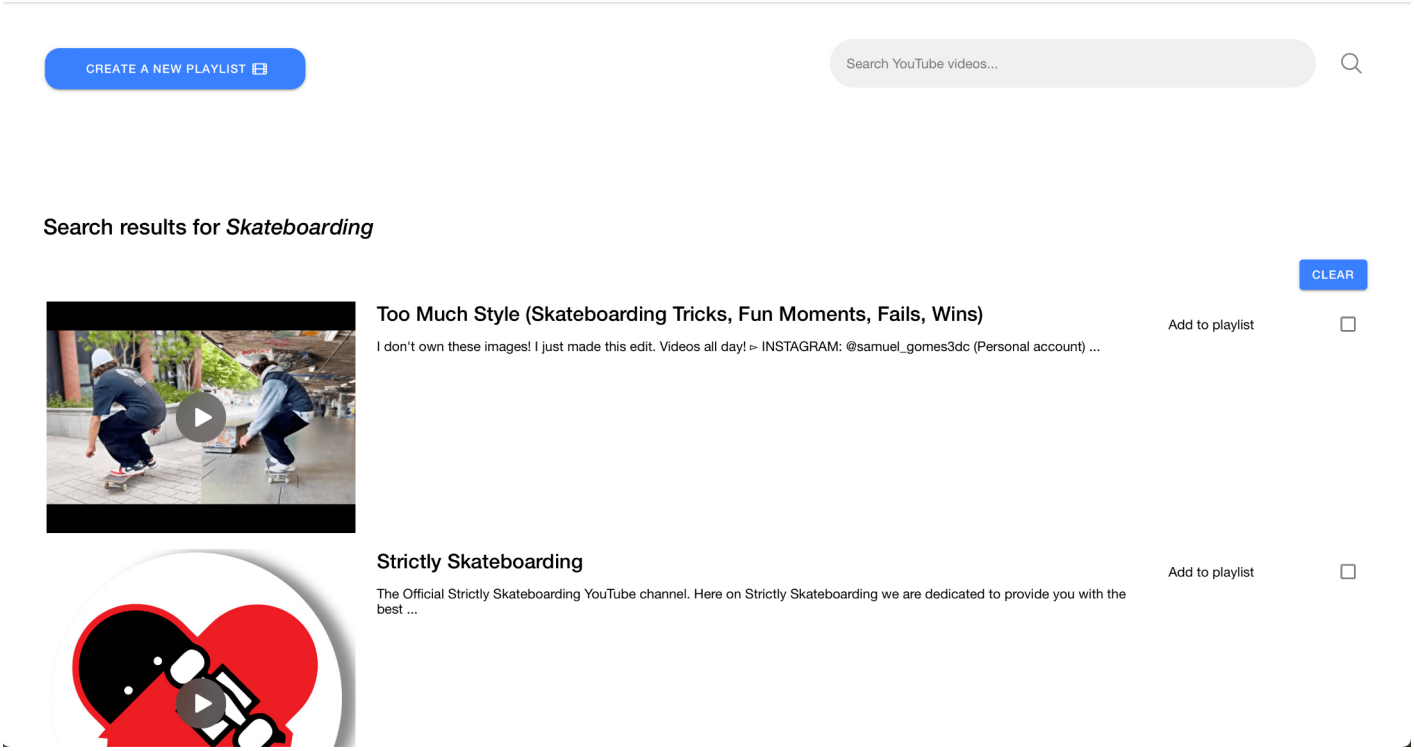
By the end of this chapter you should have a fully functioning Ionic/MEAN app that runs on your local development environment like so:



Our application UI is minimal and makes use of various Ionic UI components as well as custom Angular components to manage data input and display.

Notice a pattern here? This is the same approach we have used throughout prior projects that we have developed and covered…if it ain't broke, don't fix it!

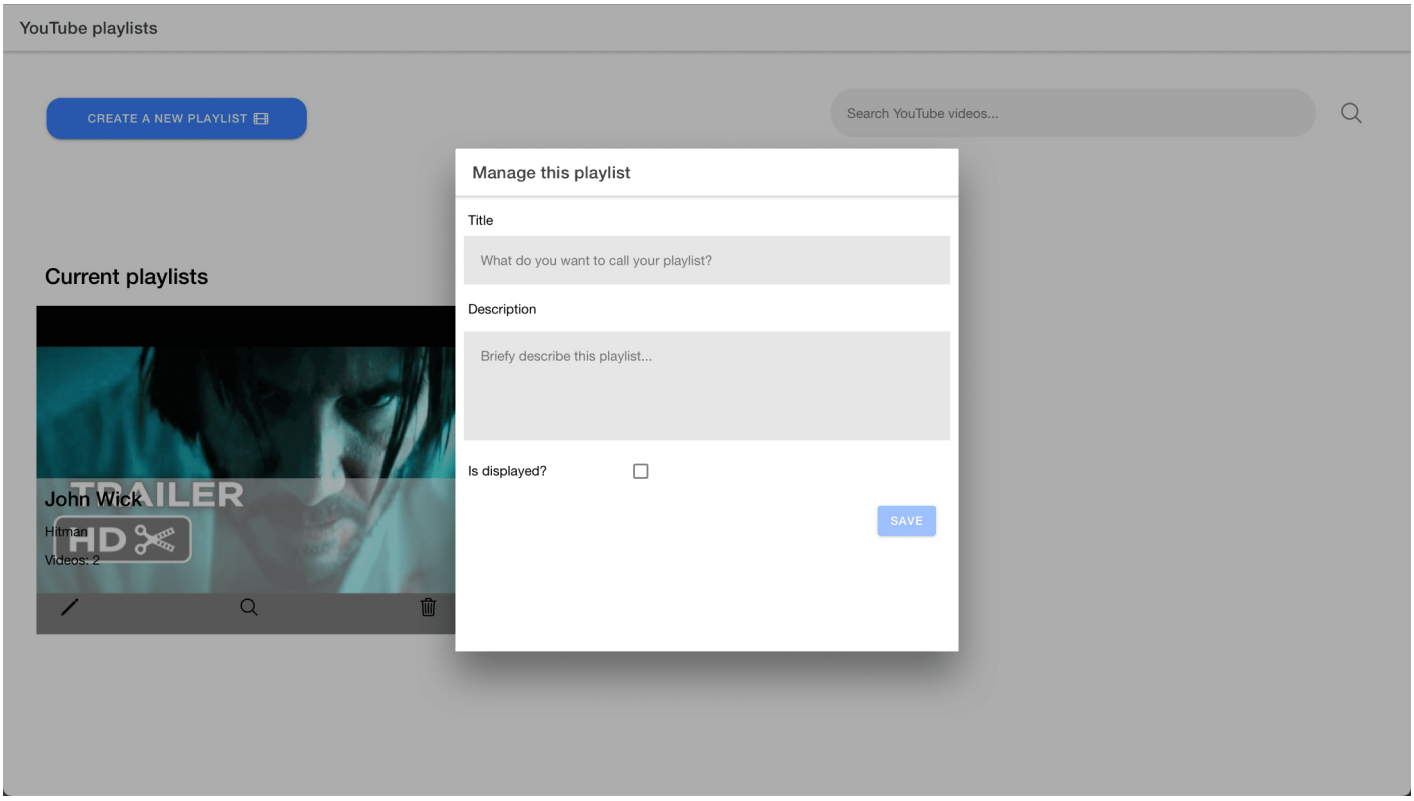
YouTube video searches are performed using the search entry bar and all returned results (assuming they exist for the supplied search term) are displayed using the Ion Grid component:



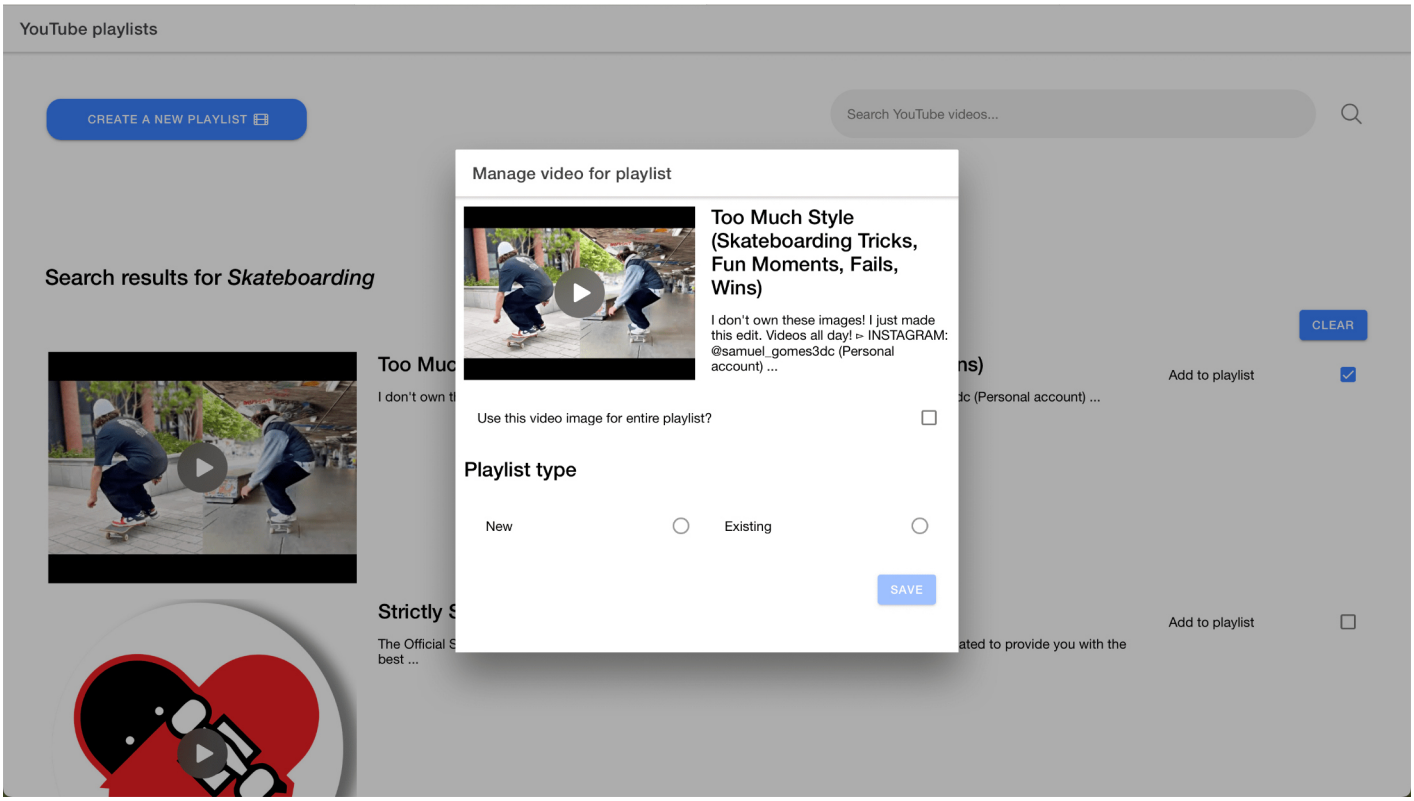
Video playlists can be created in one of 2 ways:

- Independently using the **Create a new playlist button** at the top-left hand side of the page
- When selecting a YouTube video search result to add to a playlist

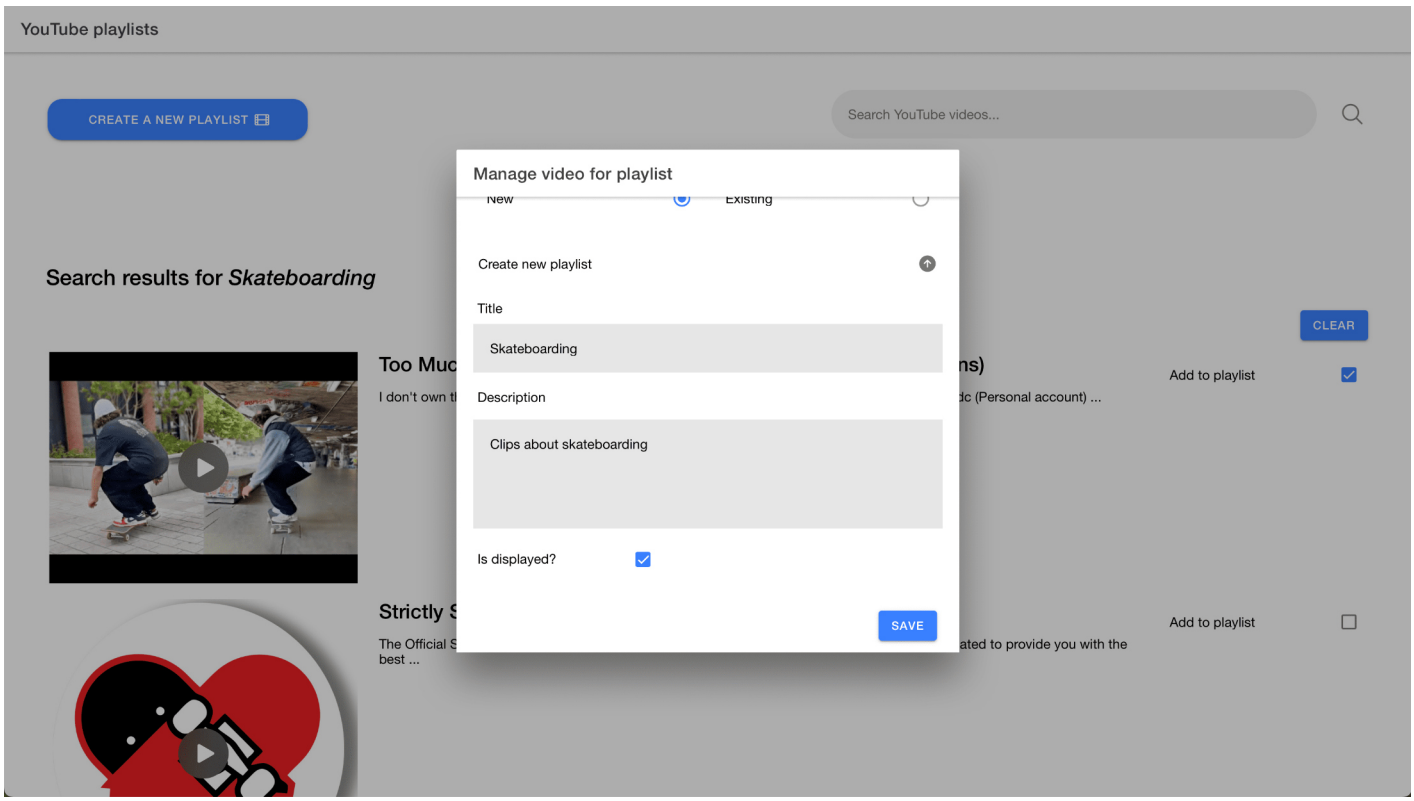
The first method to create playlist simply presents the following form displayed within an Ionic ModalController window:



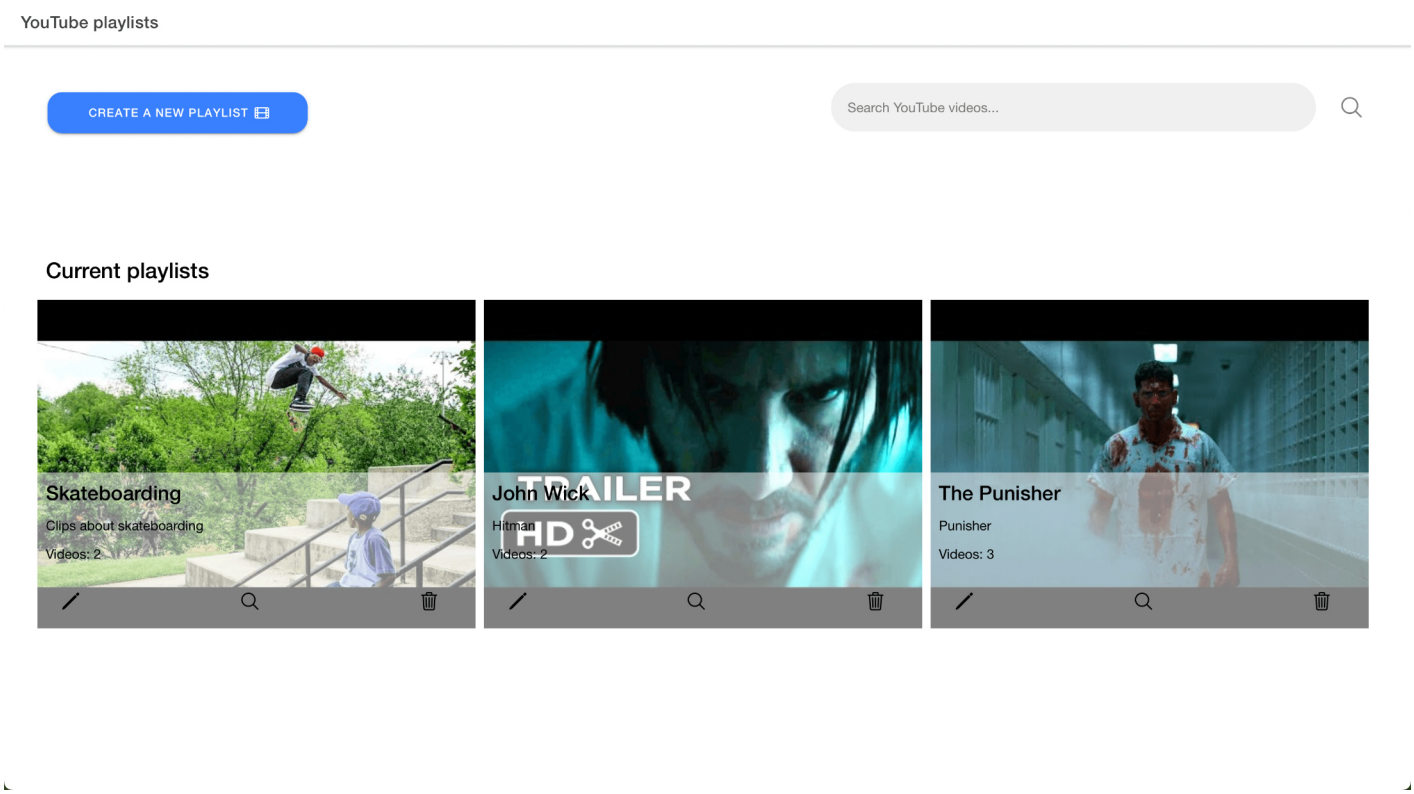
Our second method of creating a new playlist - when selecting a YouTube video search result - displays the selected item in an Ionic ModalController window and allows the user to select New from the **Playlist type** options that are presented:



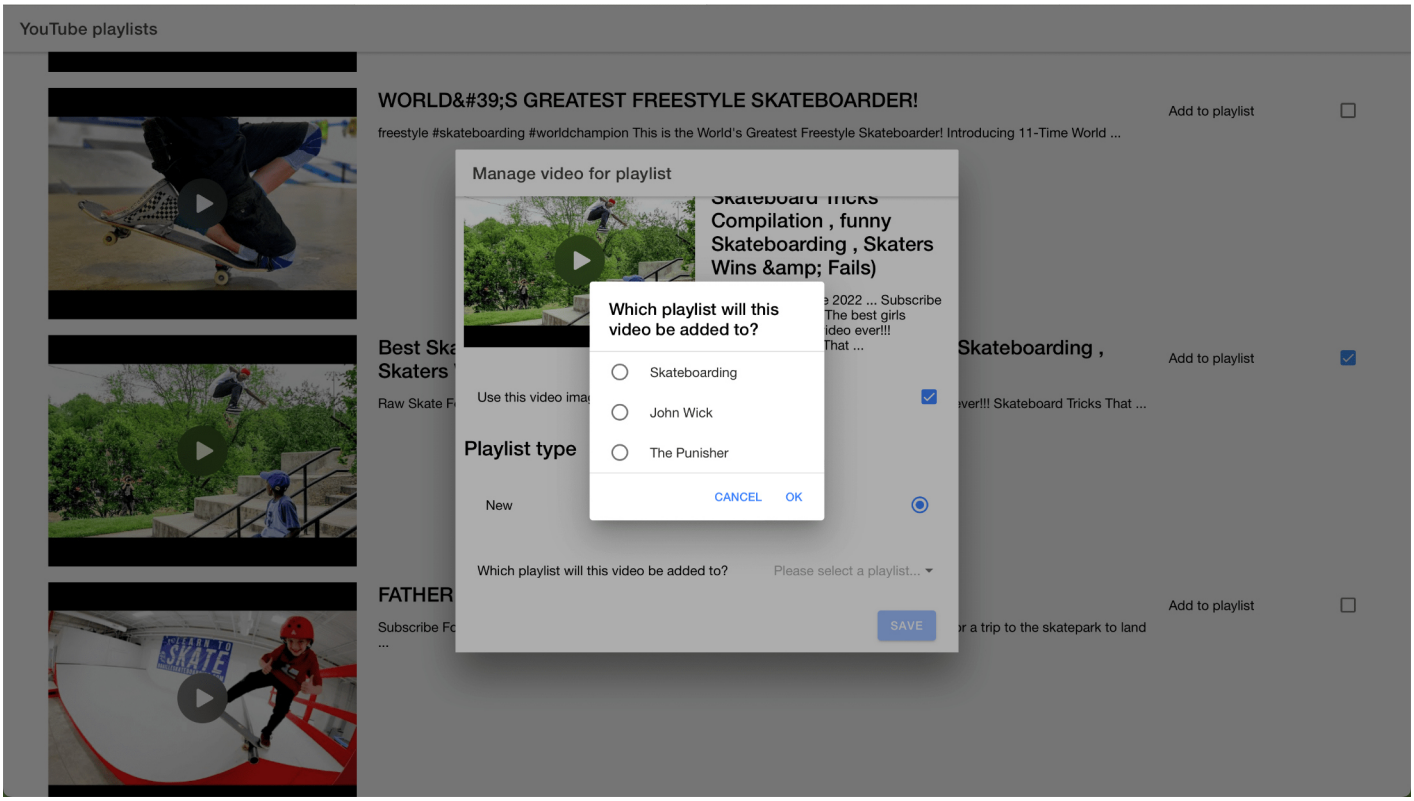
Which reveals the same form - used in our first method of creating a playlist - this time within an Ion Accordion component:



Once the completed form has been submitted and saved as a new document in MongoDB our newly created playlist is displayed on the application's home page as the first entry in all of the saved playlists:

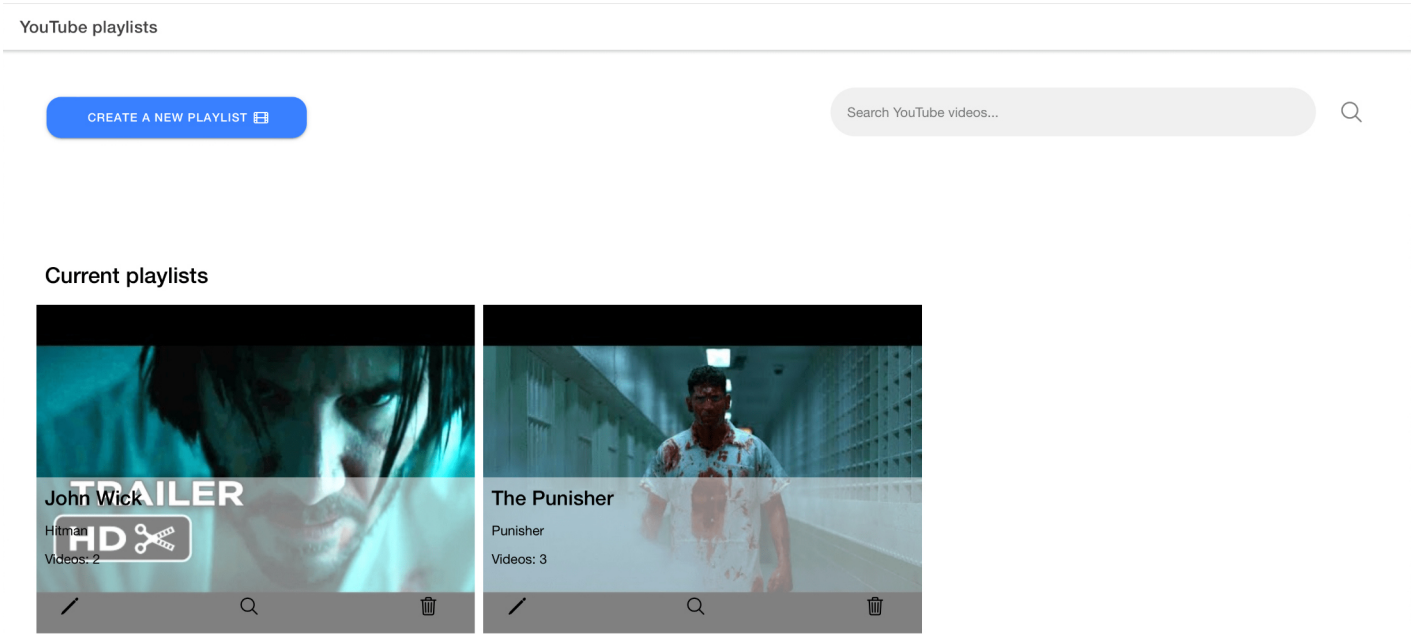


If the user had selected an existing playlist from the radio buttons options in the modal window (when assigning the selected search result entry) then this is what they would be presented with instead:

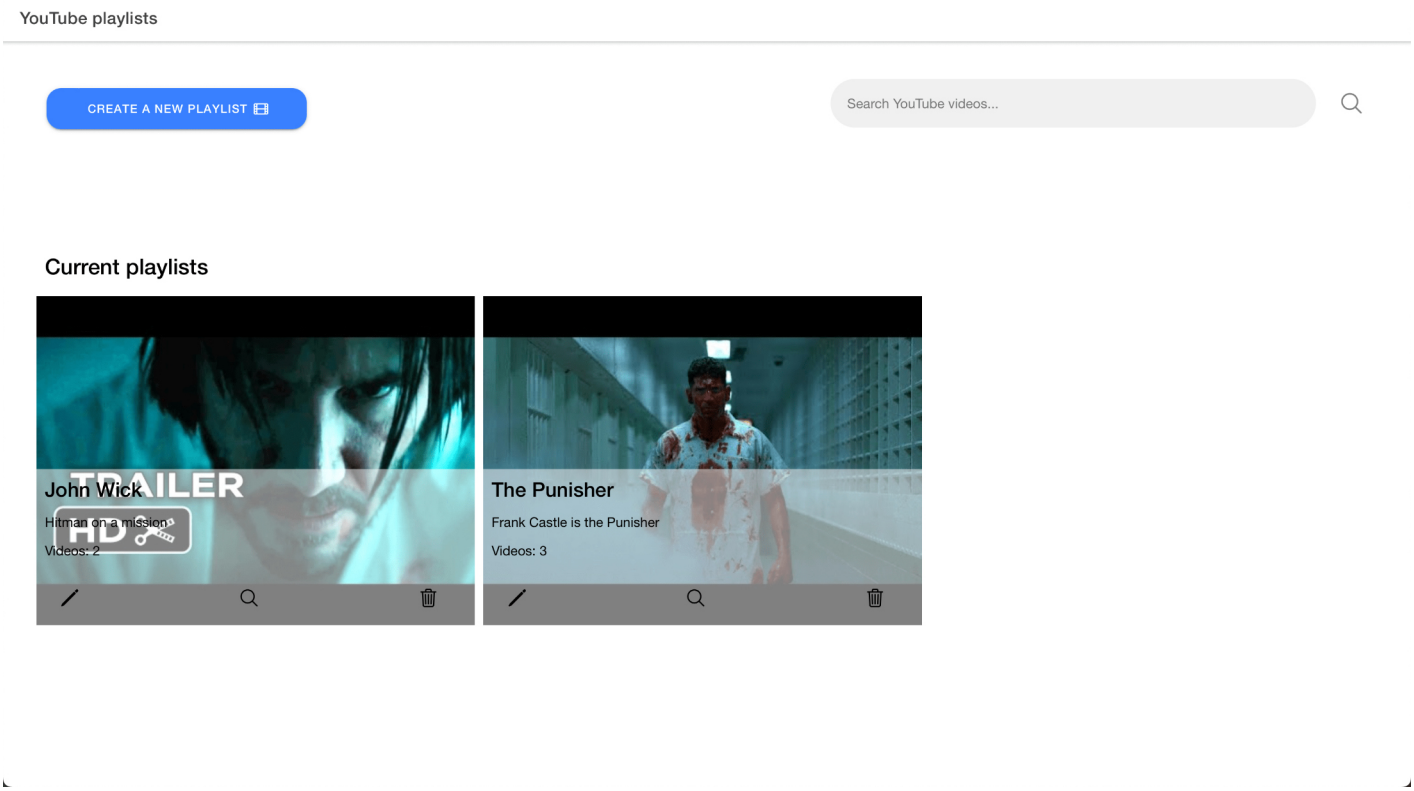


Playlists can be edited, viewed and removed using the following icons situated at the bottom of each playlist listed on the **HomePage** component view:

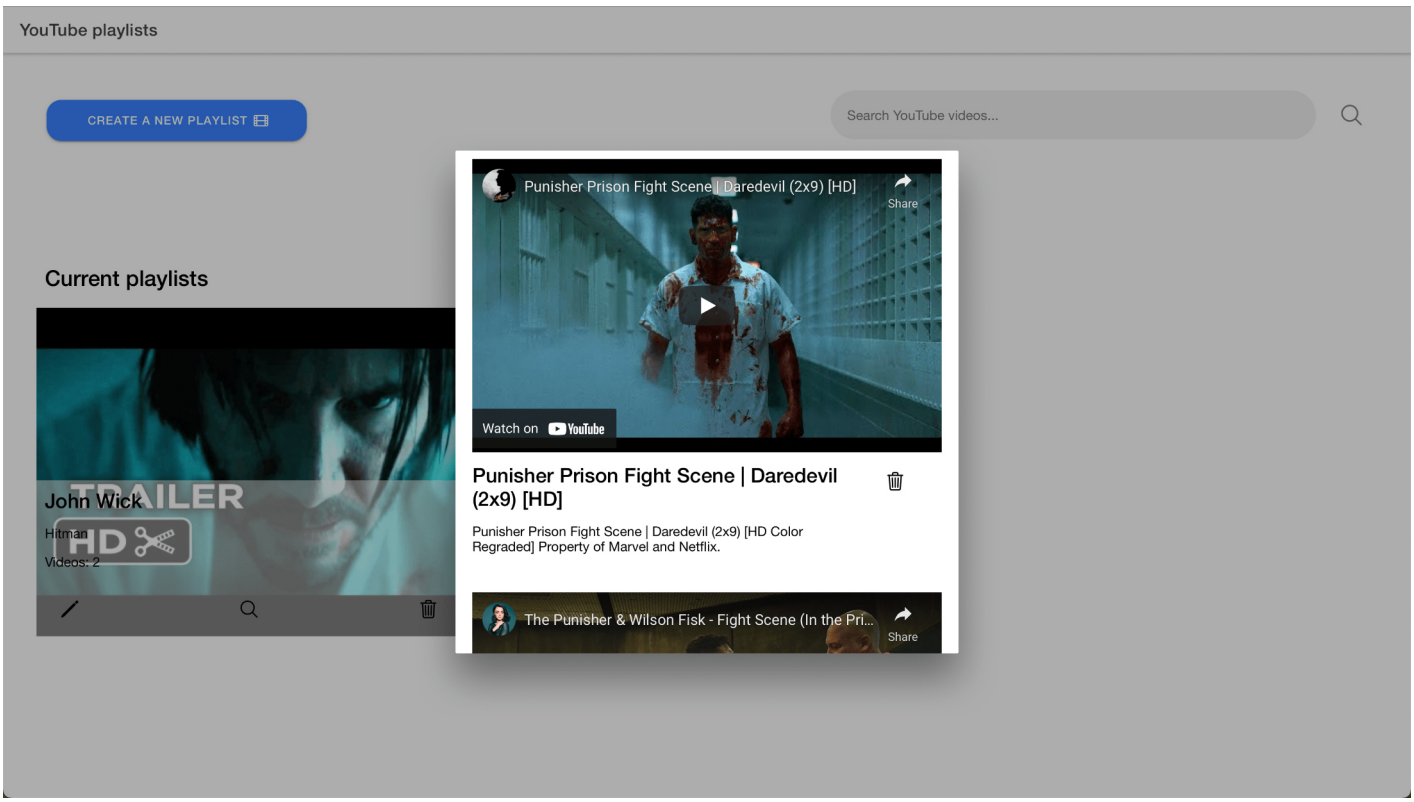
- **Pencil** - Edit the selected playlist
- **Search** - View the selected playlist (and any videos that it may contain)
- **Trash** - Delete this playlist (and all videos that it may contain)



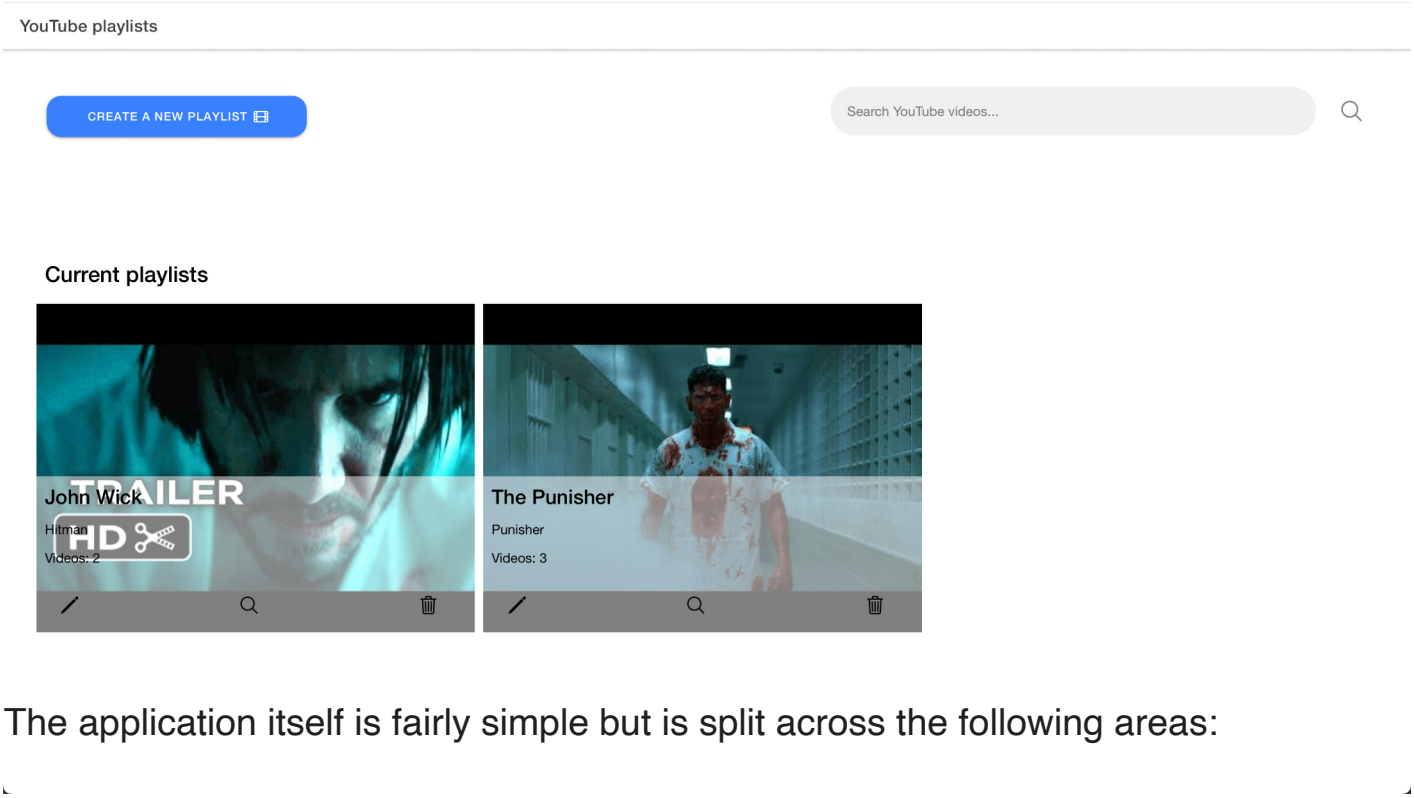
Editing a playlist publishes the same form that we used to create a playlist within an Ionic ModalController window (with those fields pre-filled with saved data for that selected playlist):



Viewing a playlist displays the image associated with that playlist (if one has been selected by the user), the title and description for that playlist along with any associated videos displayed underneath this:



Deleting a playlist simply updates the UI (in addition to deleting that document from the MongoDB collection):



The application itself is fairly simple but is split across the following areas:

- Ionic frontend
- Express routing/middleware logic
- Node server
- MongoDB instance

Let's start by creating the frontend aspect of our application, install the necessary modules and generate the different component/services that we require.

After we complete this step we'll focus on creating/configuring the Node, Express and MongoDB backend aspect of the application.

Laying the Ionic foundations

Open up your command line terminal of choice, navigate to a preferred location on your computer and start with creating a new directory titled **app**.

Within this **app** directory create a new ionic project named **ionic-youtube** and, once completed, generate the necessary Angular services, custom components and TypeScript interfaces before finishing off with installing the required npm packages:

```
mkdir app
cd ./app
ionic start ionic-youtube blank --type=angular
// Select NgModules from the menu options prompt
cd ./ionic-youtube
ionic generate interface interfaces/playlist
ionic generate interface interfaces/video
ionic generate component components/create-playlist
ionic generate component components/manage-videos-for-playlist
ionic generate component components/view-playlist
ionic generate module components/shared-components
ionic generate service services/data-change-listener
ionic generate service services/playlists
ionic generate service services/utilities
ionic generate service services/youtube
npm install --save ngx-pagination
```

Let's quickly cover the following components, services and interfaces we have just generated for this project so we are familiar with their intended purpose/function:

- **interfaces/playlist** - defines the expected structure for our playlist data
- **interfaces/video** - defines the expected structure for returned YouTube videos in search results
- **components/create-playlist** - Allows playlists to be created for the application
- **components/manage-videos-for-playlist** - Allows selected video from YouTube search results to be added to a playlist (existing or one that is created on-the-fly)
- **components/view-playlist** - Allows details of the playlist and all assigned videos to be viewed
- **components/shared-components/shared-components.module.ts** - Feature module to allow custom components to be exported for use in other parts of the application
- **services/data-change-listener.service.ts** - cross-component communication for notification of changes to data
- **services/playlists.service.ts** - manages the creation/amendment/removal and retrieval of playlists

- **services/utilities.service.ts** - supplies utility methods for use throughout the application
- **services/youtube.service.ts** - Performs YouTube video search by keywords and returns any matching results

We'll return to these later and cover them in more depth but first let's turn our attention to the server side of the application.

Routing, middleware and databases

The backend for our application consists of NodeJS for our server, Express for routing and middleware management and MongoDB for data storage and retrieval.

Before we go any further we need to ensure these are installed on and, where necessary, configured for our system - starting with our web server.

If you haven't already installed the latest version of node on your development environment then download the installer for your particular system and follow the installer steps before proceeding.

If this is already in place then we can move swiftly onto our next step…

Installing & configuring MongoDB

The simplest and quickest way to install MongoDB on Linux/Mac OS is using the package manager tool Homebrew.

If you don't have Homebrew currently installed on your Mac OS system then run the following from the command line to rectify this:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Alternative installation options for Homebrew can also be found here.

Once Homebrew has been successfully installed simply run the following command to install the Community Edition of MongoDB to your system:

```
brew install mongodb
```

Be sure to follow the steps listed here to successfully run the installed MongoDB software.

With MongoDB now installed on our machine let's explore the underlying database and concepts before taking a little sidestep and exploring a GUI application that we can use to interact with created databases.

Data modelling according to Mongo

MongoDB, as explained earlier, is a document database solution that stores data in the form of JSON object-based documents. If you're familiar with SQL databases then it might be a little 'odd' at first to understand how document-oriented databases tackle data storage.

Simply put a MongoDB database instance is structured around the use of collections, documents and fields.

Broadly speaking a collection can be thought of in similar terms to a SQL database table; it is used to store category specific data (I.e. user details, restaurant locations, film genres etc).

A document, if we continue our comparison with SQL terms, can be seen as the individual 'row' of data within a collection.

Finally, a field in a document can be thought of as a column in a SQL table. Similar to their SQL column counterparts a field is able to be defined by particular data types (such as String, Boolean, Date etc) and indexes where required.

It's important to note that each document can have different fields applied to it if and where required, which makes document-oriented databases like MongoDB, radically different from traditional SQL databases with their strict schemas.

For the purposes of this tutorial we're going to maintain a strict schema for the documents within our MongoDB database instance using a node package called

Mongoose which will allow us to map objects to data (more on this later).

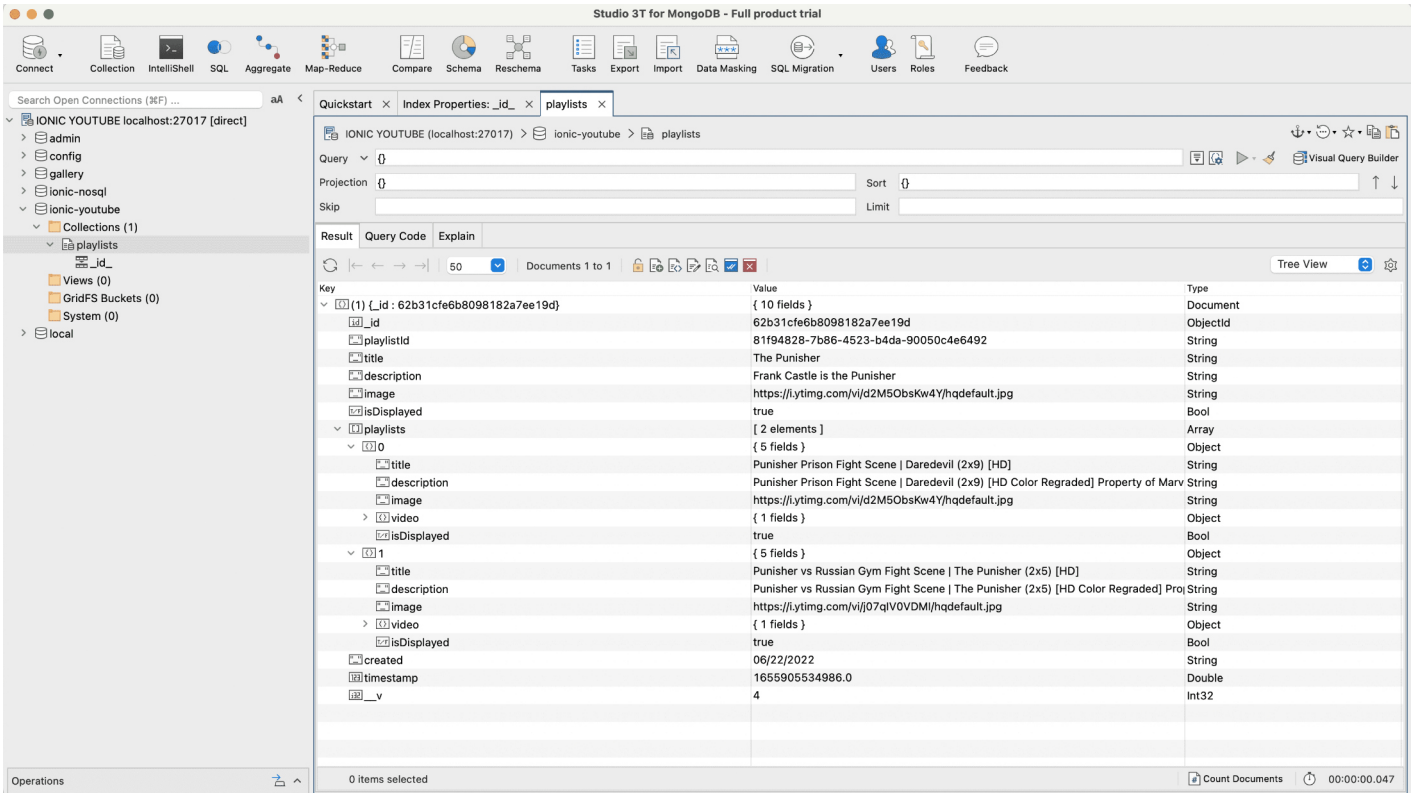
With this conceptual map in place let's quickly explore a GUI tool that we can use to interface with MongoDB databases.

Studio 3T

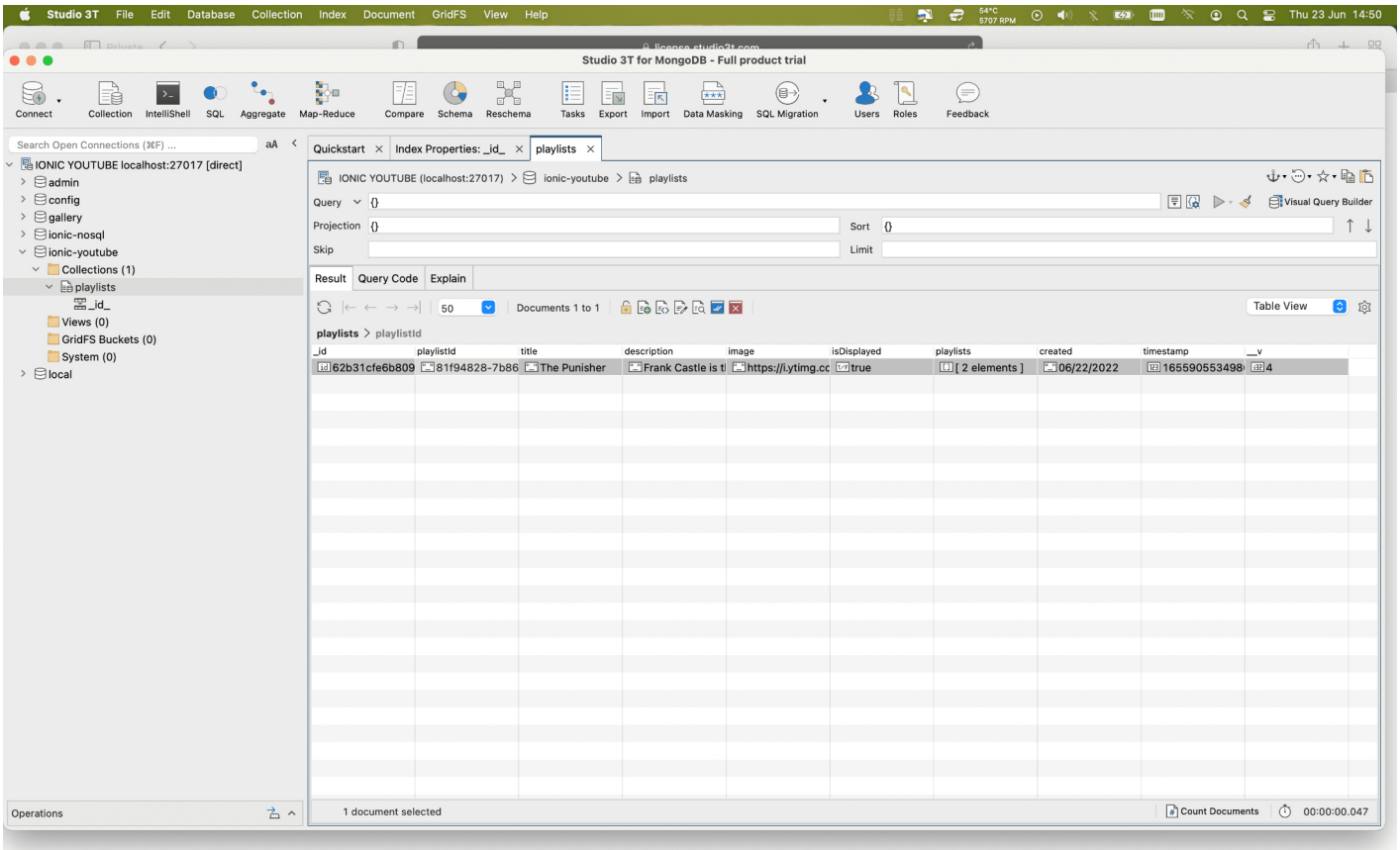
Formerly known as Robo 3T Studio 3T is a free, open-source, cross platform GUI tool (Windows, Mac OS and Linux) for managing local MongoDB databases.

Once MongoDB has been installed on your system you can simply interface with the software through the Studio 3T tool instead of the command line (which although not inefficient can be tedious at times to work with - no matter how hardcore a developer you might be it's hard to argue against the benefits of a GUI when it comes to data management).

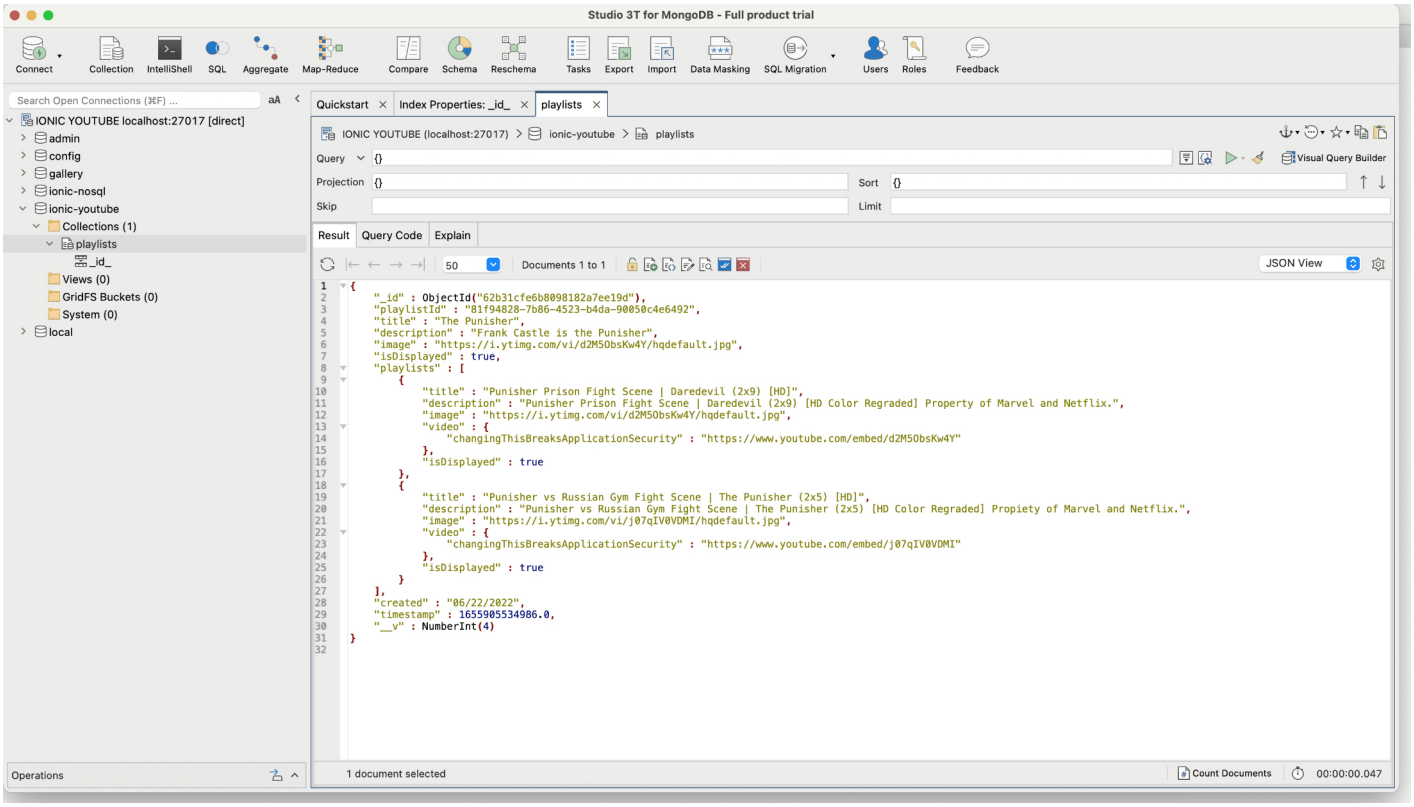
For example, you can view a collection's documents in tree view:



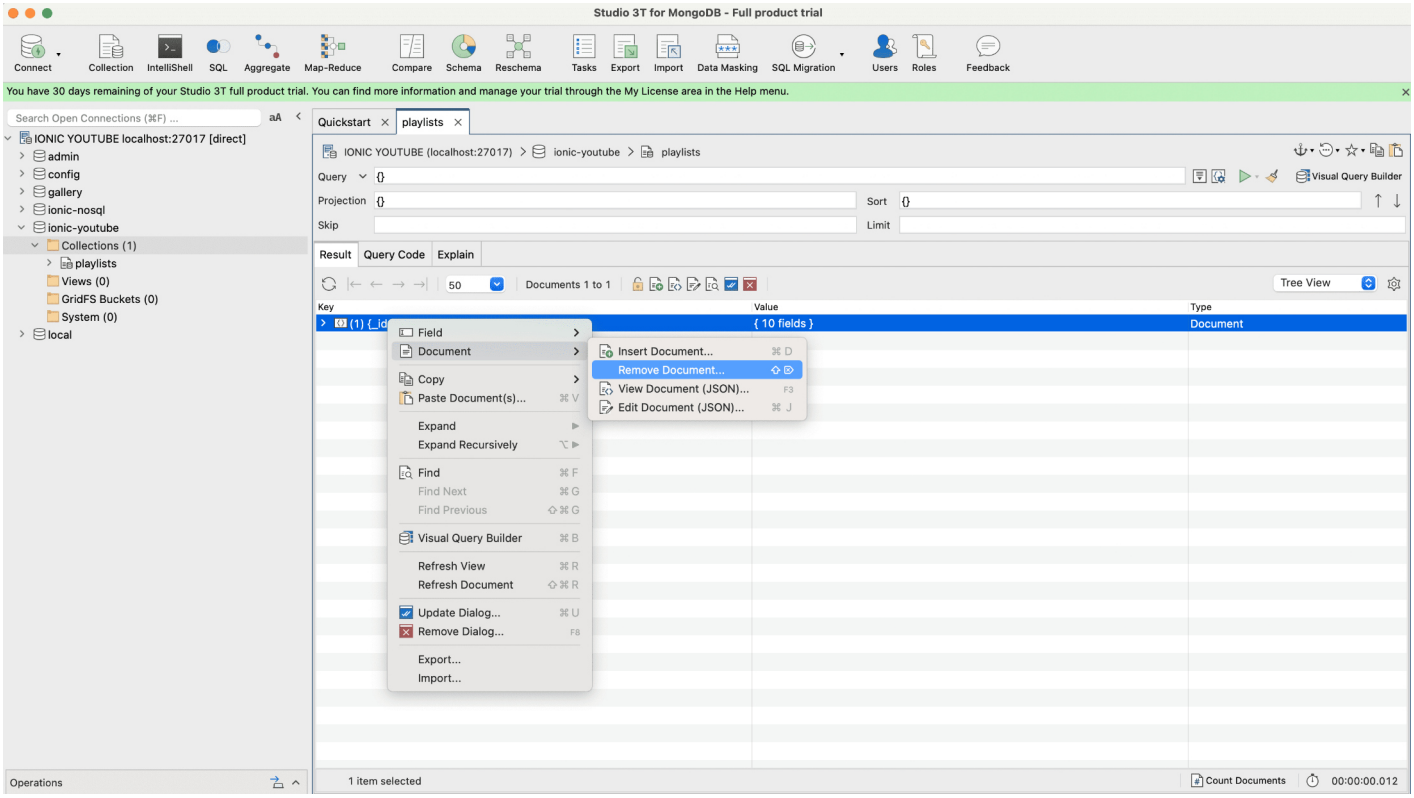
Alternatively, you can choose to view a collection's documents in a table view instead (which, depending on the number of fields - and their types/quantity of data, doesn't always make for the most effective UI to engage with said data):



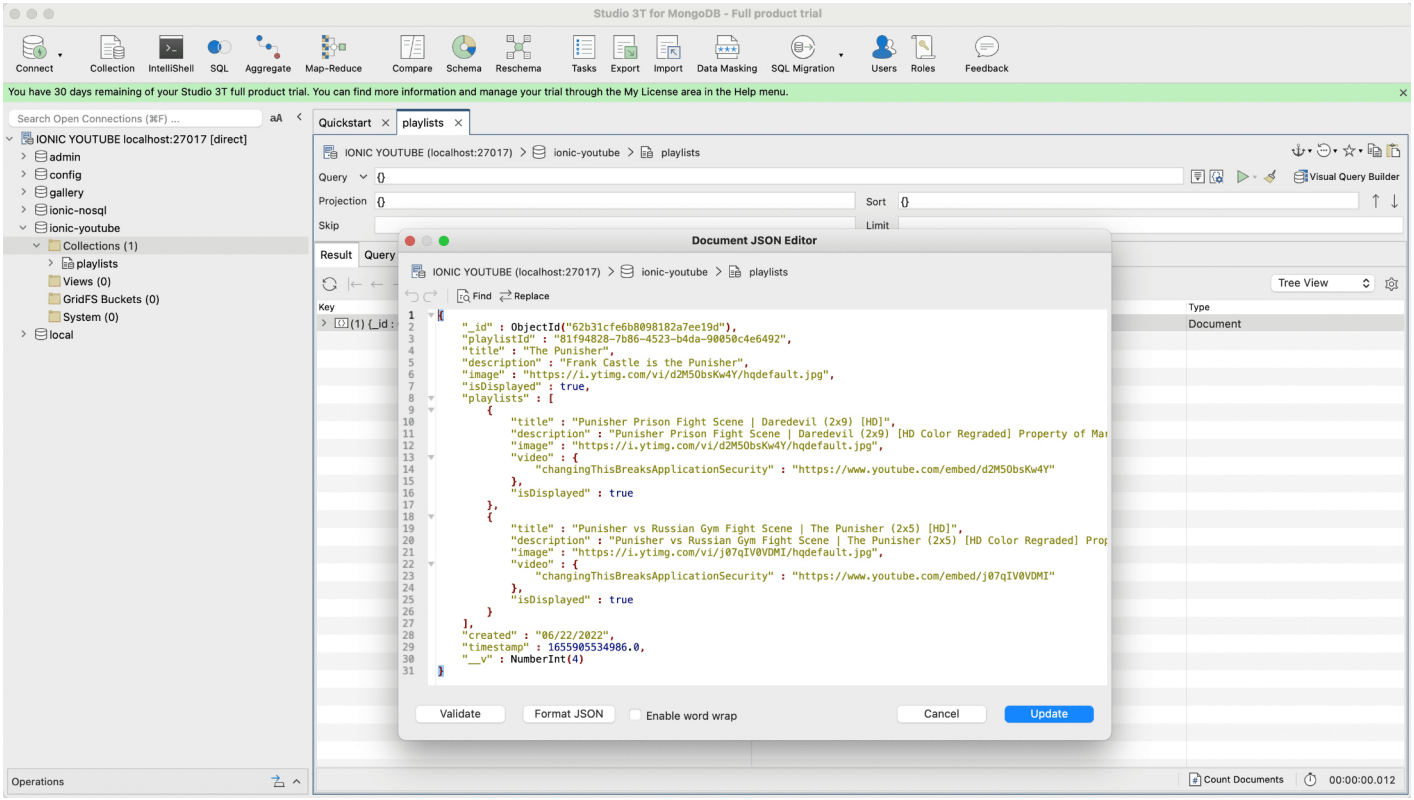
Or maybe just opt for a document view instead (which is my personal favourite as it directly mirrors working with code):



Documents can be individually managed by right-clicking on the selected document to present a contextual menu listing the available options:



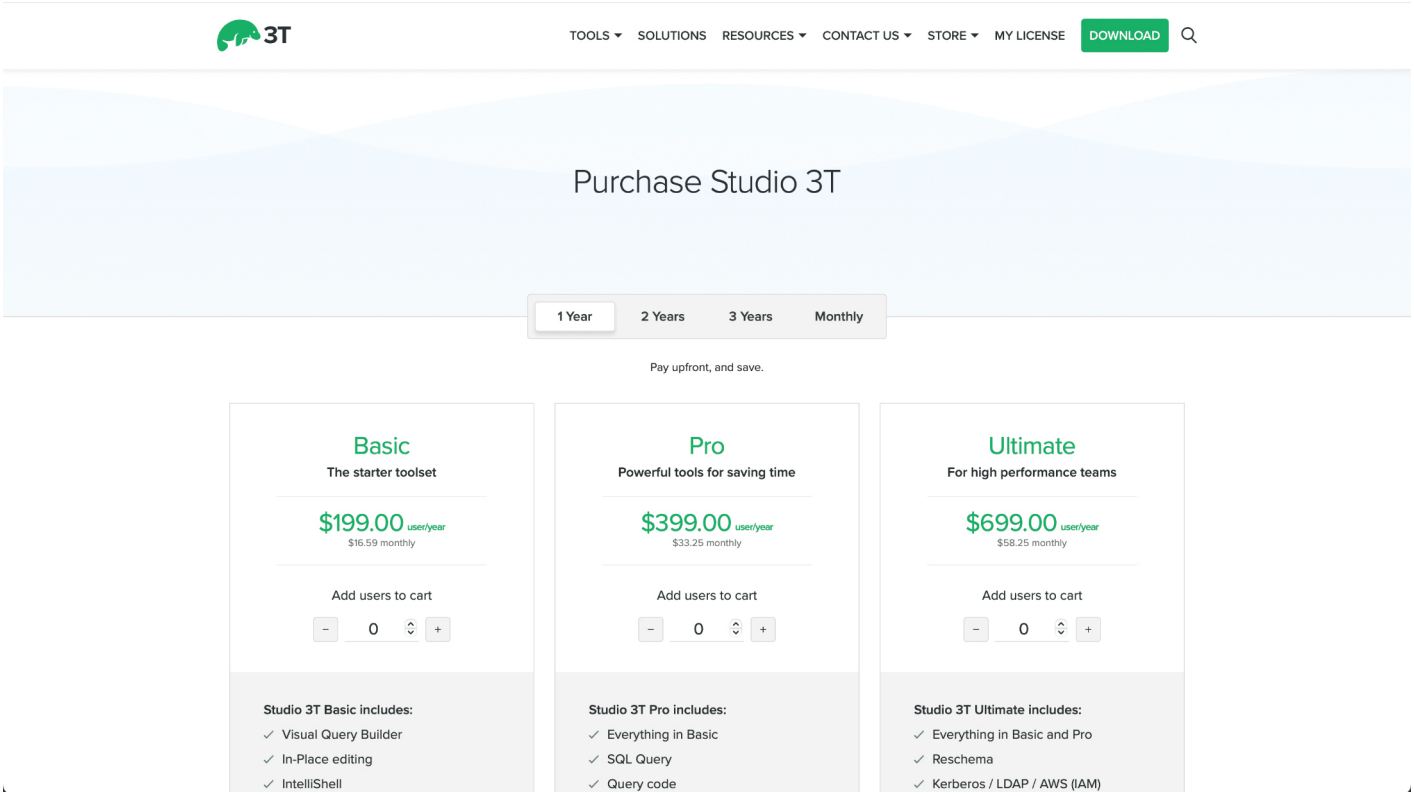
The selected document is then opened within a modal window and able to be edited in-situ:



As you've no doubt gathered from the above screen captures Studio 3T makes interfacing with and managing MongoDB database collections and documents quick and easy to accomplish. With such a well designed, intuitive and user friendly interface the software is a must-have for any developer working with MongoDB (and

it certainly beats using the command line - although Studio 3T does offer that facility too).

One word of caution though - the Studio 3T price tag is quite hefty unless you are working almost exclusively with MongoDB (and generating decent income from your particular products/services to justify purchasing a full license):



In the context of this project we'll only be using Studio 3T to test the results of managing the MongoDB data with our Ionic/Node application (but it's a tool you might want to consider using in your own MongoDB projects - if you aren't already).

Setting the server side foundation

Remember earlier when we generated the Ionic project with its Angular components, services & interfaces that we did so within a sub-directory that we created with the name of **app**?

The **app** sub-directory is the container for all of the front-end of our application.

Now we need to create the container for the server-side of our application.

Navigate to the parent directory that contains the **app** sub-directory and create

another sub-directory with the name of **server**.

You should now have the following directory structure (I have named the parent containing directory **ionic-youtube** just to identify the overall name of the project):

```
ionic-youtube
    |
    |-- app
    |
    |-- server
```

Backend setup

Open your system CLI and navigate to the root of the **server** directory before running the following command to generate a node project:

```
npm init
```

With each CLI prompt that is subsequently displayed (these are used to generate the **package.json** file for the project) simply accept all the defaults before running the following packages:

```
npm install --save body-parser
npm install --save cors
npm install --save express
npm install --save mongoose
npm install --save uuid4
npm install --save nodemon
```

As you are no doubt aware node has a large ecosystem of packages that can be added to projects so let's take a moment to walk through each of the above and discuss why they are installed for this project.

- **Express** - a web application framework for node that provides features such as routing (defining navigation paths within the application), HTTP utility methods

End of preview