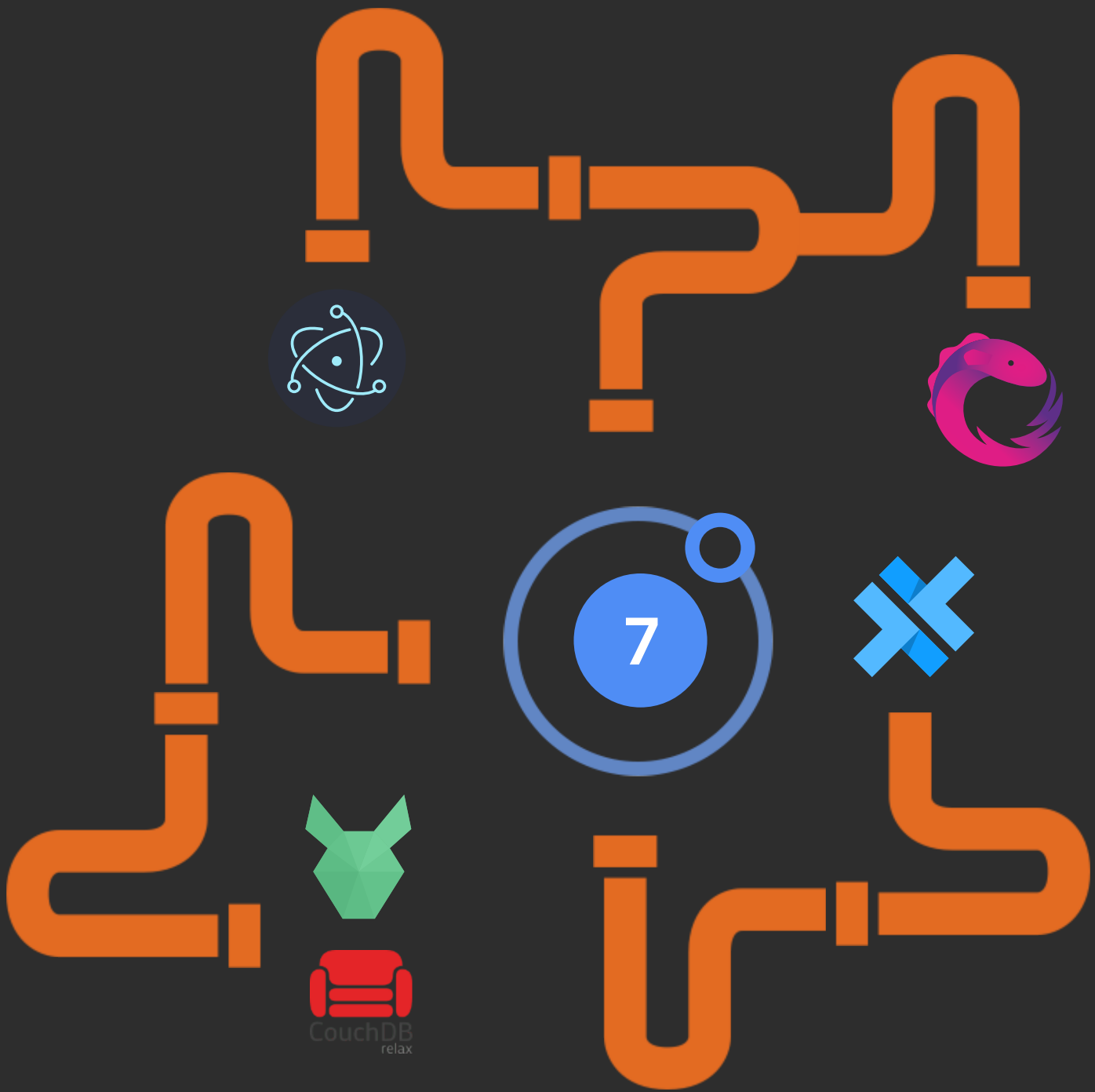
Mastering Ionic

Working with PouchDB



James Griffiths

Mastering Ionic - Working with PouchDB

Saints at Play Limited

Copyright © 2017 by Saints at Play Limited

Notice of rights

All rights reserved. No part of this book may be reproduced or transmitted in any form by any means (electronic, mechanical, photocopying, recording or otherwise) without the prior written permission of the author.

If you have a copy of this e-book and did not pay for it you are depriving the author and publisher of their rightful royalties. Please pay for your copy by purchasing it at Leanpub.

For all enquiries regarding obtaining permission for book reprints and excerpts please contact the author at Leanpub.

Notice of liability

The information contained within this book is distributed on an “As Is” basis without warranty.

While every precaution has been taken in the preparation of the book and its supplied computer code, neither the author, Saints at Play Limited (including its agents, associates and any third parties) nor the publisher shall be held liable to any person or entity for any loss or damage howsoever caused, or alleged to be caused, whether directly or indirectly, by the content and instructions contained within this book and/or the supplied computer code or by the computer hardware and software products described within its pages.

Trademarks

This e-book identifies product names and services known to be trademarks, registered trademarks, or service marks of their respective holders which are used purely in an editorial fashion throughout this e-book.

In addition, terms suspected of being trademarks, registered trademarks, or service marks have been appropriately capitalised, although neither the author, Saints at Play Limited (including its agents, associates and any third parties) nor the publisher cannot attest to the accuracy of this information.

Use of a term in this book should not be regarded as affecting the validity of any trademark, registered trademark, or service mark. Neither the author, Saints at Play Limited (including its agents, associates and any third parties) nor the publisher are associated with any product or vendor mentioned in this book.

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 5

Glossary 12

Databases - A short summary 16

PouchDB - Database API abstraction layer 29

Application development 127

Case Study - PouchDB Jigsaw 143

In closing 293

Author biography 294

Project downloads 295

Introduction

Thank you for purchasing this digital copy of Mastering Ionic: Working with PouchDB.

My goal with this ebook is to guide you through working with the frontend database PouchDB to seamlessly integrate this within an Ionic/Angular front-end (whether this is to be delivered via the browser or as an Electron desktop app).

We start with exploring database concepts and terminology - for both SQL and NoSQL databases before progressing onto 2 separate projects featuring applications powered by Ionic/PouchDB.

Within each project that we cover we'll work with the simple, but powerful, PouchDB API and leverage its features 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
- Custom Search JSON API
- Google's Programmable Search engine
- rxJS library methods
- ElectronJS.

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 of PouchDB - including every API method that the database offers then this simply is NOT the book for you (you can access the online documentation for that purpose).

What is covered

This book provides detailed information on the core essentials of PouchDB and working with its most commonly used API methods, performing CRUD related operations and how to integrate data from PouchDB 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 PouchDB (and all of its core features) is simply not possible (as the book would have to be at least twice as large to accommodate this…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 as well as deploying projects to Electron 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 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 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

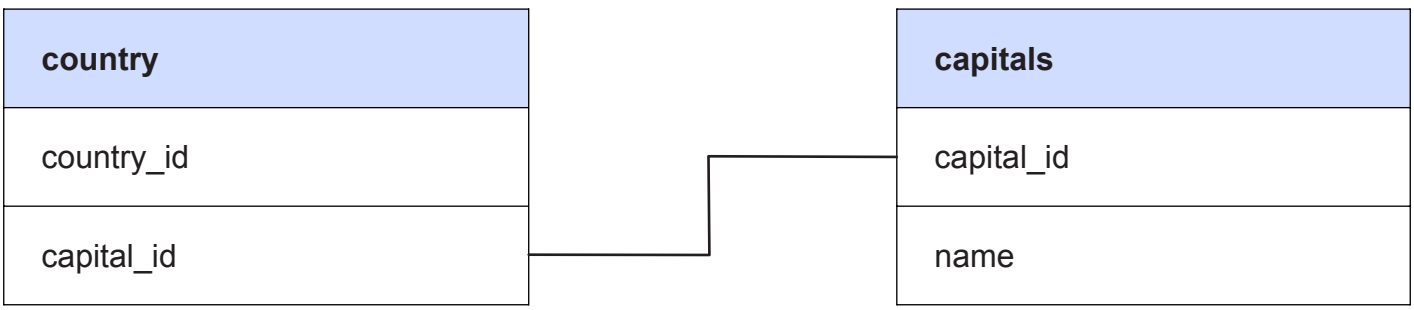
Relationships

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

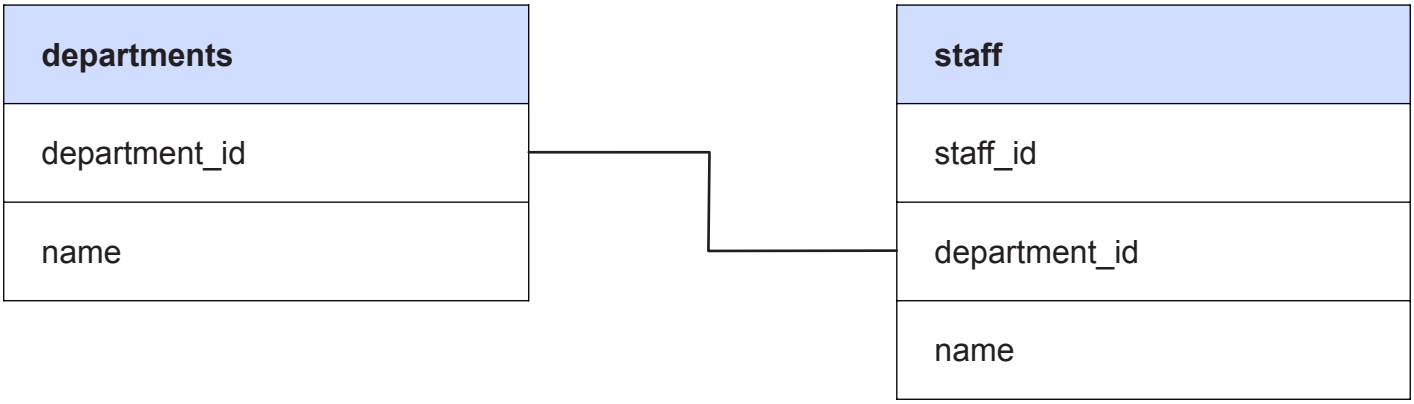
- **Primary key** - A table column where each record has a unique value
- **Foreign key** - A table column whose values references the primary key of another table

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

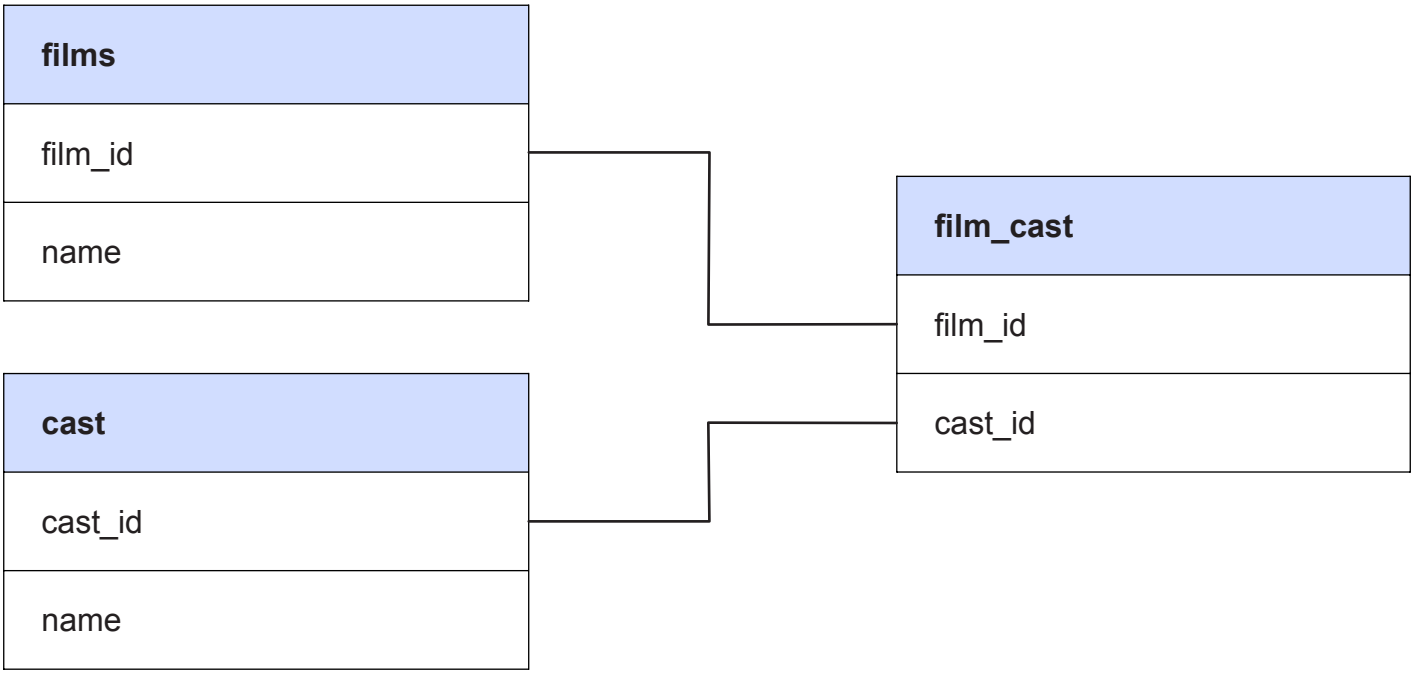
A **One-to-one** relationship consists of a single record on each side - for example, the relationship between a country and its capital city (as a country will only ever have one capital):



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

PouchDB

Database API abstraction layer

PouchDB is an open-source, JavaScript, NoSQL, client-side database that runs in the browser, using the browser's underlying storage mechanism - typically IndexedDB or, if this is unavailable/unsupported, defaulting to WebSQL instead.

This essentially makes PouchDB a sort of abstraction layer, providing developers with a single API through which the software then determines the appropriate database storage mechanism to use.

As you can imagine this helps simplify the process of handling data within our web and mobile apps as developers don't have to choose between writing for WebSQL or IndexedDB - PouchDB simply manages that in the background (note: as of PouchDB 7.0 WebSQL is not supported by default but must be enabled through use of the PouchDB WebSQL adapter).

Given the fragmented nature of web browsers (even with modern browser support there are still organisations/individuals running legacy software which doesn't always play nice - or even at all - with modern web technologies) there's always that question…..can I use this?

With PouchDB you're covered from the following browsers/platforms upwards:

- Firefox 29+ (Including Firefox OS and Firefox for Android)
- Chrome 30+
- Safari 5+
- Internet Explorer 10+
- Opera 21+
- Android 4.0+
- iOS 7.1+
- Windows Phone 8+

The only time where this will clearly be an issue is where browser versions prior to the above need to be supported (thankfully such instances should be very rare indeed…that said let's hope that you never find yourself working with a client where you still have to still support IE6 of all browsers!)

If you are unfortunate enough to find yourself in a situation where you DO need to

support older browsers PouchDB provides some guidance on measures you might want to consider.

Given that we'll be developing with Ionic/Angular we're supported as far using PouchDB is concerned.

Great…so what does PouchDB look like?

The PouchDB API

As a NoSQL database solution PouchDB provides a robust and intuitive JavaScript API that developers can use to manage their front-end database interactions.

The API syntax should look somewhat familiar to JavaScript/TypeScript/Ionic (delete as applicable here) developers (comments have been added for guidance):

```
// Create a PouchDB instance
const db = new PouchDB("name-of-database-here", {size: 50})

// Add a document to the database
const snippetToAdd = {
  _id: new Date().toISOString(),
  content: title
  };

db.put(snippetToAdd, function callback(error, result) {
  if (!error) {
    console.log('Whoo! We successfully posted an entry to the database!');
  } else {
    console.log('Dang! We messed up somewhere');
    console.dir(err);
  }
});

// Remove a document from the database
const snippetToRemove = {
```

```
    _id: id,
    _rev: rev
};

db.remove(snippetToRemove , function callback(error, result) {
    if (!error) {
        console.log('Whoo! We successfully removed an entry from the database!');
    } else {
        console.log('Dang! We messed up somewhere');
        console.dir(err);
    }
});
```

Pretty self-explanatory right?

Some of you might be recoiling at the use of callbacks in the above methods but the PouchDB API (which is entirely asynchronous) allows developers to manage such asynchronous operations using any of the following approaches:

- callbacks
- Promises
- async/await

Simply take your pick and work with what suits you best!

We're not going to dwell on the syntax for the API here as we'll be covering that while we're developing our app but if you're feeling a little impatient you can always visit the online docs for further information.

In a nutshell here's why you might want to consider using PouchDB in your next Ionic project:

- Provides a consistent, robust and intuitive database abstraction API
- Adapters for different front-end storage technologies (WebSQL, LocalStorage and In-memory)

- Wide range of plugins for extending/augmenting PouchDB capabilities
- Can be used locally/remotely (across HTTP)
- Synchronises with CouchDB on server

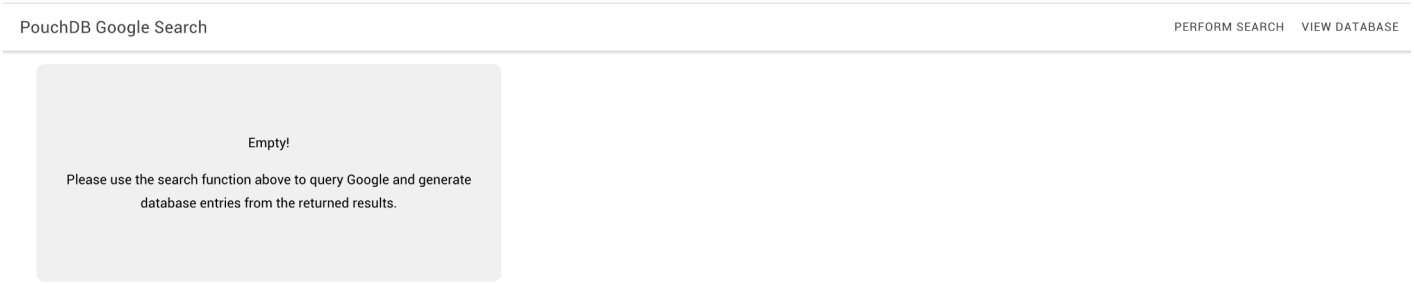
With our brief walkthrough of the basics of PouchDB covered let's now take a look at what we're going to be developing over the course of this chapter.

Our PouchDB/Ionic app

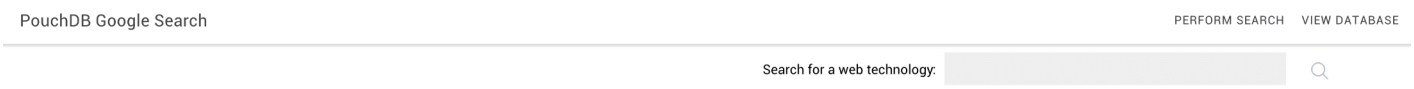
Working with Ionic (and the underlying Angular framework) we'll implement PouchDB as a database abstraction layer to develop a simple but functional application that manages/performs the following:

- Integrates the Google Custom Search API
- Allows user to perform searches for Ionic/front-end related technologies
- Returned results can be saved to PouchDB at the user's discretion and full control
- Allows PouchDB CRUD operations to be performed with selected search entries

In its default state our application will appear as follows:



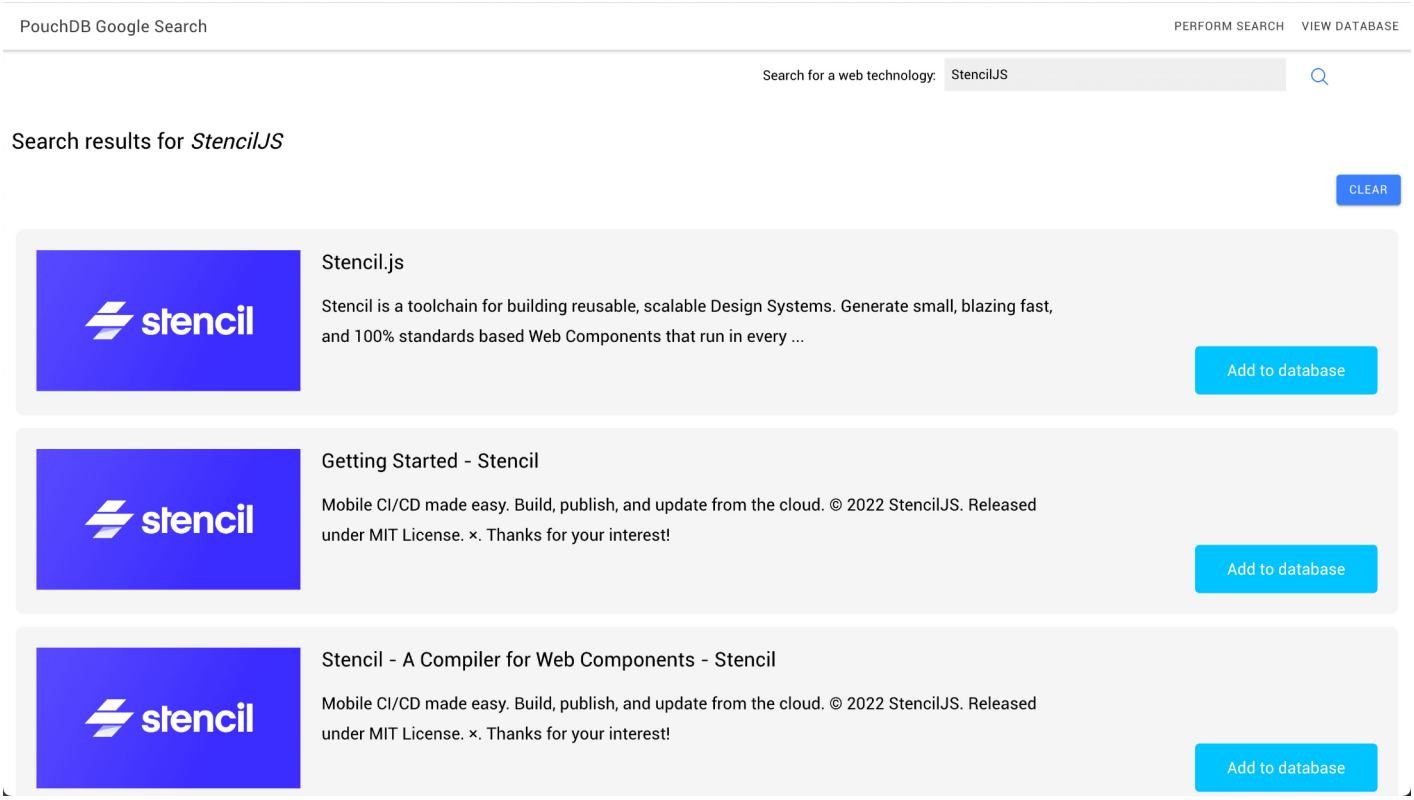
We can select the perform search button in the top-right hand corner of the screen to display the search input field where we can enter a technology related search term:



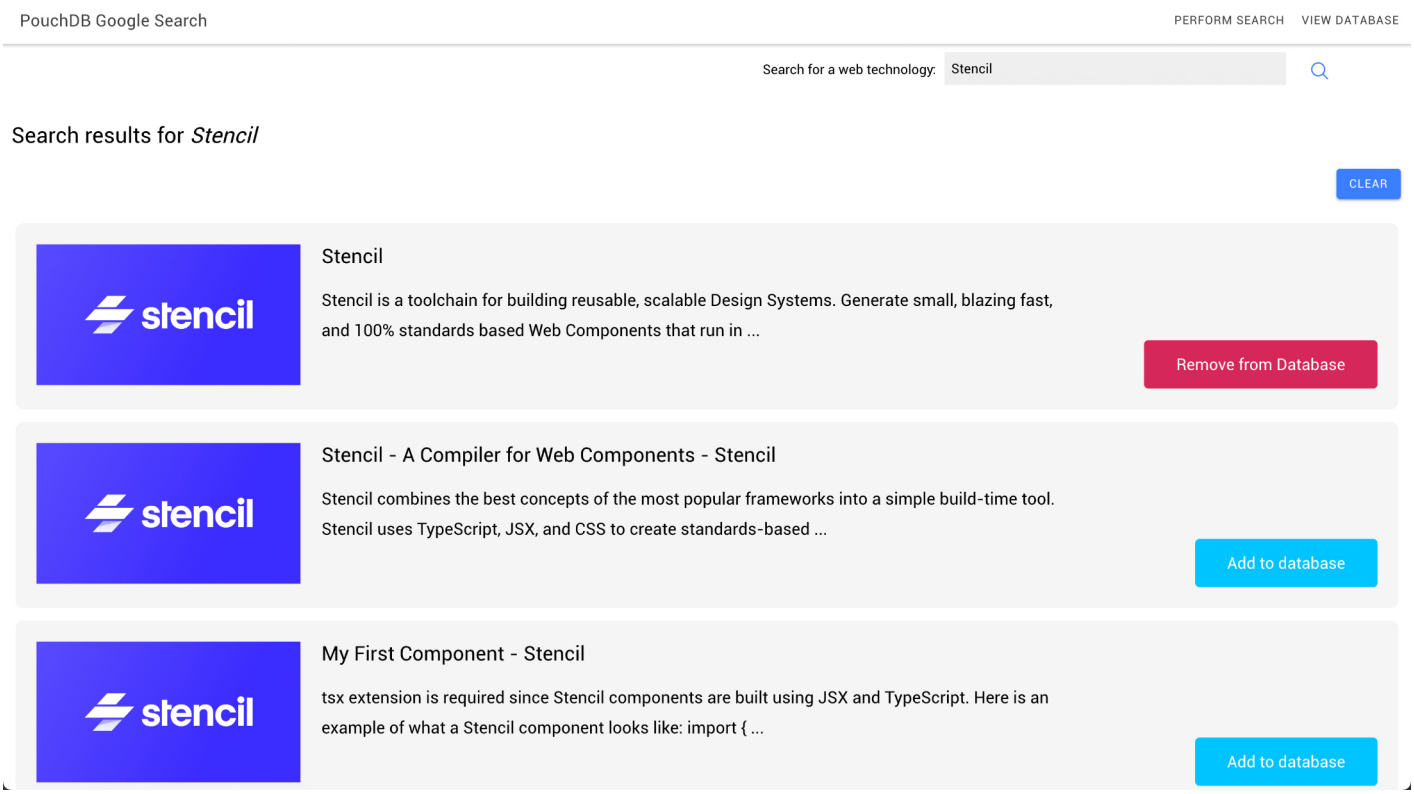
As you can see our search screen - in its default state - is quite minimal.

This changes when results are displayed from the Google Search API.

Results returned for our submitted search term are displayed in their own custom Angular component - allowing the user the option to individually save a result to PouchDB - or to clear all of the returned results:

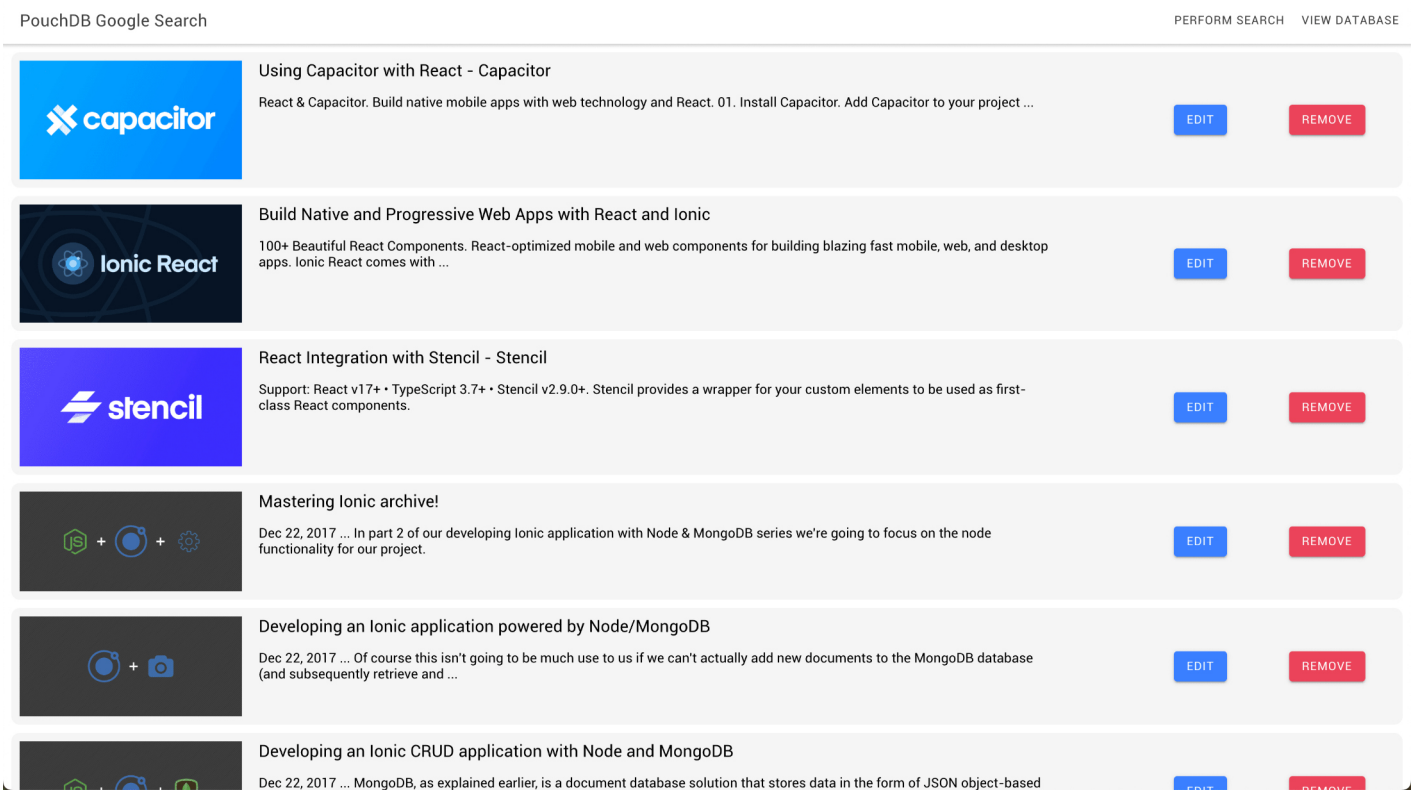


When a user selects a specific search result the button changes colour and the text now displays the instruction **Remove from Database** (allowing the user to instantly identify which record(s) have been selected as well as to be able to reverse their selection if they so wish):

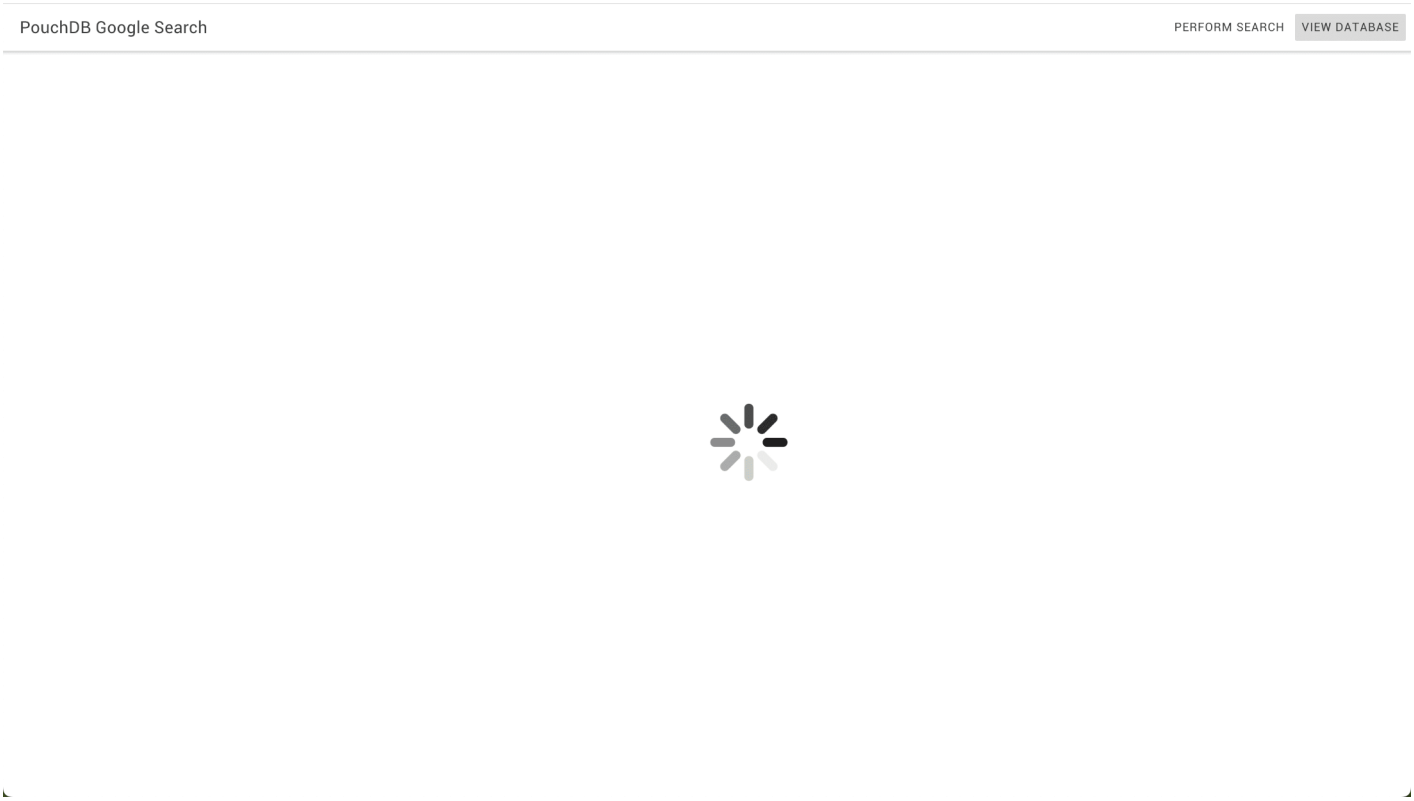


Selecting the **View Database** button at the top right-hand corner of the page allows all existing PouchDB documents to be asynchronously rendered to the screen.

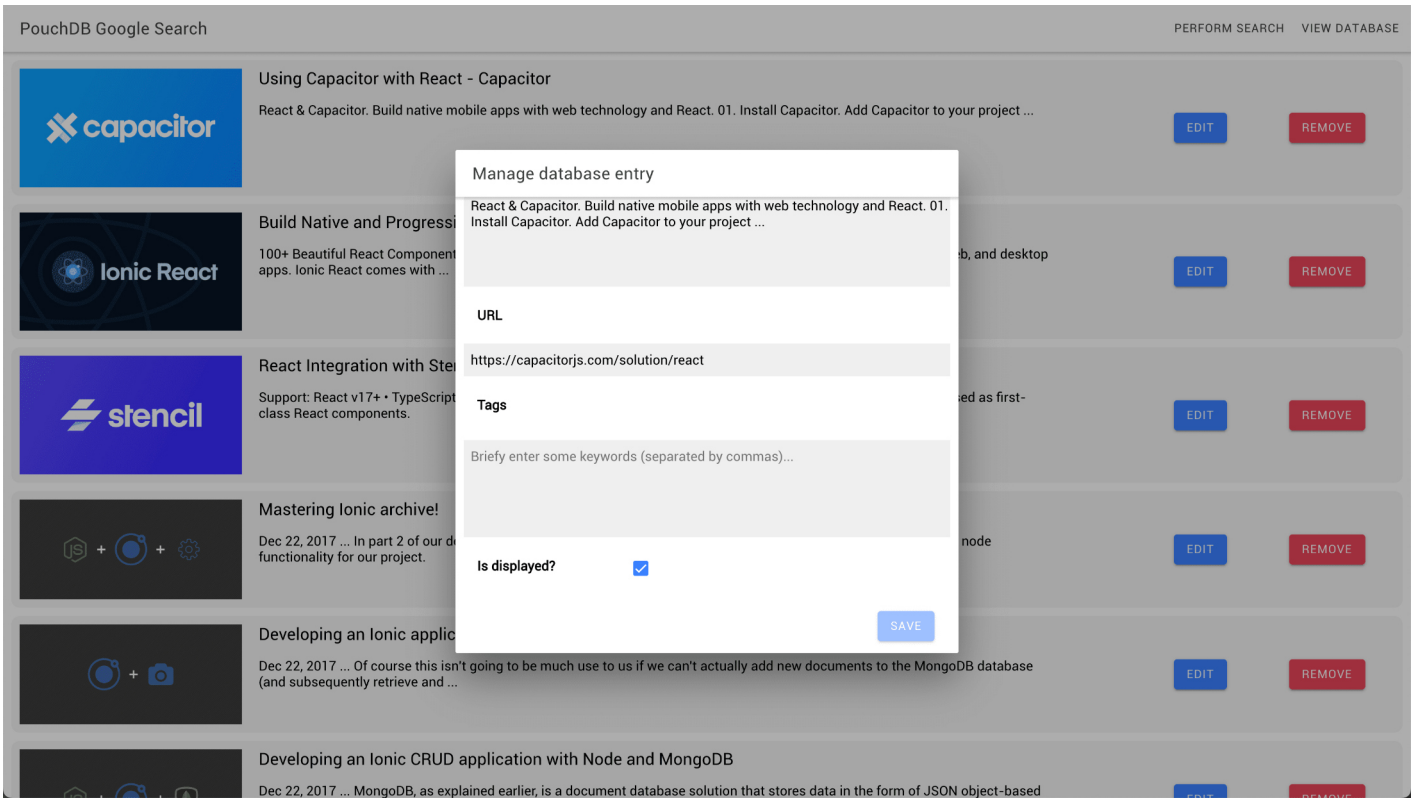
Each displayed listing allows the user to amend or remove that from PouchDB:



As our data is loaded asynchronously (for both database entries and search results) we use a loader animation to signal to the user that this process is underway:



Clicking on the edit button for a rendered database entry displays the editable fields for that record in an Ionic ModalController window (with Angular ReactiveForms validation implemented for the form):



Our application is pretty simple in scope but covers the fundamentals of working with PouchDB as well as demonstrating how different technologies can work together in the context of an Ionic application.

Now that we know what we'll be developing let's create the basic Ionic project skeleton (and then focus on setting up the Custom Search API for Google searches).

Laying the foundations

Assuming that you have all the necessary system/software prerequisites (mentioned in our getting started chapter) in place open up your system command line and issue the following instructions to create a new Ionic project, install the necessary software modules and generate the required page components, interfaces, modules and services:

```
ionic start pouchdb-technologies blank --type=angular
// Select NgModules from the menu options prompt
cd ./pouchdb-technologies/
npm install --save pouchdb
npm install --save-dev @types/pouchdb
npm install --save pouchdb-browser
npm install --save-dev @types/pouchdb-browser
ionic generate interface interfaces/result
ionic generate service services/pouch-db-manager
ionic generate service services/google-search
ionic generate service services/utilities
ionic generate service services/data-change-listener
ionic generate component components/search-result
ionic generate component components/display-entry
ionic generate component components/manage-entry
ionic generate module components/shared-components
```

Let's quickly cover the files we have generated here and their purpose.

Starting with our **app/interfaces** directory we have the following single interface:

- **interfaces/result** - will contain the "contracts" for the expected structure of the returned Google Custom API search results and saved database entries

Our services are contained within their own dedicated **app/services** directory and will be responsible for managing the "heavy lifting" for certain functionality:

- **services/pouch-db-manager** - will make use of the PouchDB API to provide database configuration logic and CRUD methods for the application
- **services/google-search** - supplies keywords/terms to Google Custom Search API and returns results
- **services/utilities** - "catch-all" service for generic functionality used within the app
- **services/data-change-listener** - Uses the rxJS library BehaviorSubject for cross component communication where data changes need to be subscribed (and responded) to

Our custom Angular components are located within the **app/components** directory and consist of the following:

- **SearchResultComponent** - Manages display of each returned result from the Custom Search API query
- **DisplayEntryComponent** - Manages display of each saved database record
- **ManageEntryComponent** - Allows a saved database entry to be edited/viewed

Finally, within the **app/components/shared-components** directory, we also declare an Angular feature module to allow our custom components to be exported for use throughout different areas of the application:

- **SharedComponentsModule** - Angular module that exports the custom components allowing them to be imported into other application modules where requested (and not imported across ALL of the application - as they would be if imported within the application root module - making this a more optimised approach to managing component usage)

We'll cover each of the above in further depth a little later in this chapter but first we need to set up our Custom Search API through Google.

Enabling the Custom Search JSON API

In Google's own words:

> *The Custom Search JSON API lets you develop websites and applications to retrieve and display search results from Programmable Search Engine*

programmatically. With this API, you can use RESTful requests to get either web search or image search results in JSON format.

One important point to note about this API: users can make 100 search queries per day for free (after this limit has been passed billing must be enabled).

This daily free quota gives us enough room to play, test and experiment with the API in our development projects.

There are two aspects that need to be in place to implement and use the Custom Search JSON API in your projects:

- Create and configure a Programmable Search Engine
- Create a project API key

Let's go through these step-by-step - starting with creating the Programmable Search Engine.

What's a Programmable Search Engine?
Google offers developers/organisations the opportunity to add a customisable search feature to their websites/applications to deliver results that are powered by the Google Search service.

As the name of this feature implies the Programmable Search Engine allows the user to program the content that is to be searched - whether that is from a single source (such as their own website) or multiple sources.

We'll be using the Custom Search JSON API to query our Programmable Search Engine (which, as mentioned earlier, is free to use for up to 100 queries per day) but first we need to actually create - and configure - our Programmable Search Engine.

Begin by signing into the Programmable Search Engine Control panel using your Google account or, if you don't have such an account, create a new one here.

Once logged in you will be directed to the following screen where you are invited to enter the url of the site that you wish to search.

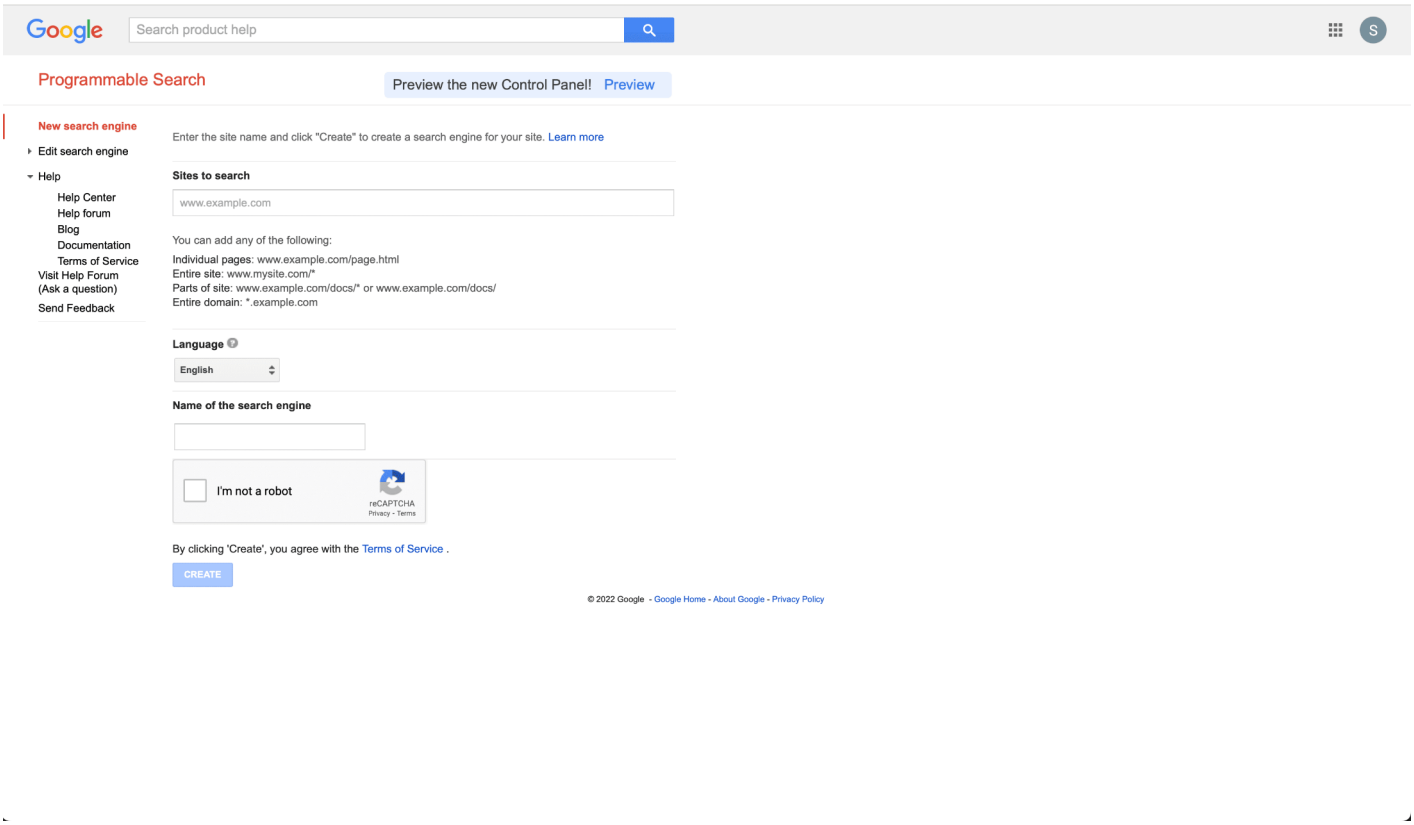
You can enter multiple URLs (one URL per new auto-generated input field) if you wish (and that is exactly what we will be doing for the purposes of this project).

You can, if you so wish, change the default language for the page layout and localisation of the search engine (useful if you happen to be providing results in Hebrew or Arabic for example).

For the purposes of this chapter I'm going to leave this set to the default English.

Finally, you need to provide a name for your search engine.

In its default state this is how the screen appears:

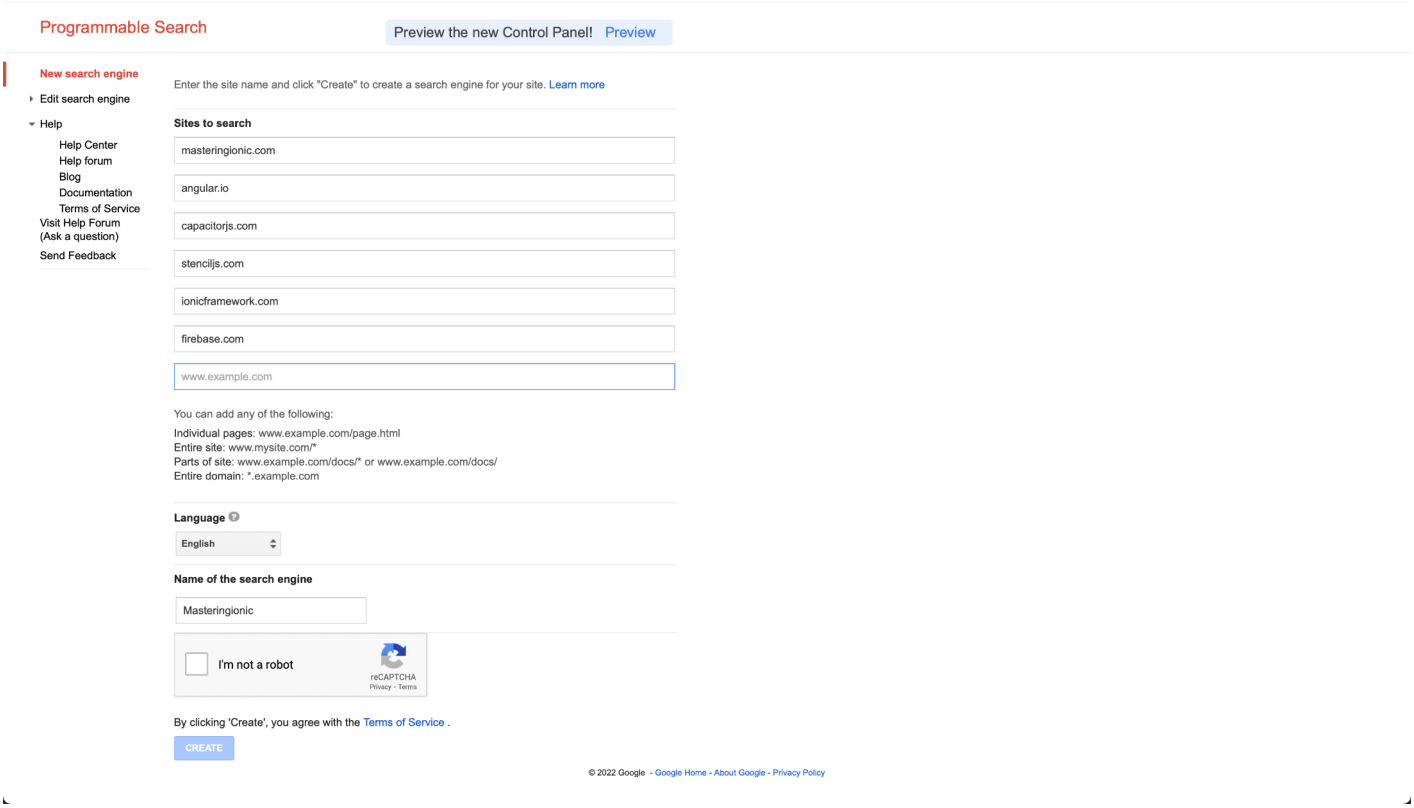


For the purposes of this project we'll be adding the following websites (if you can think of suitable alternatives or additional resources for searching then feel free to augment or completely replace this list):

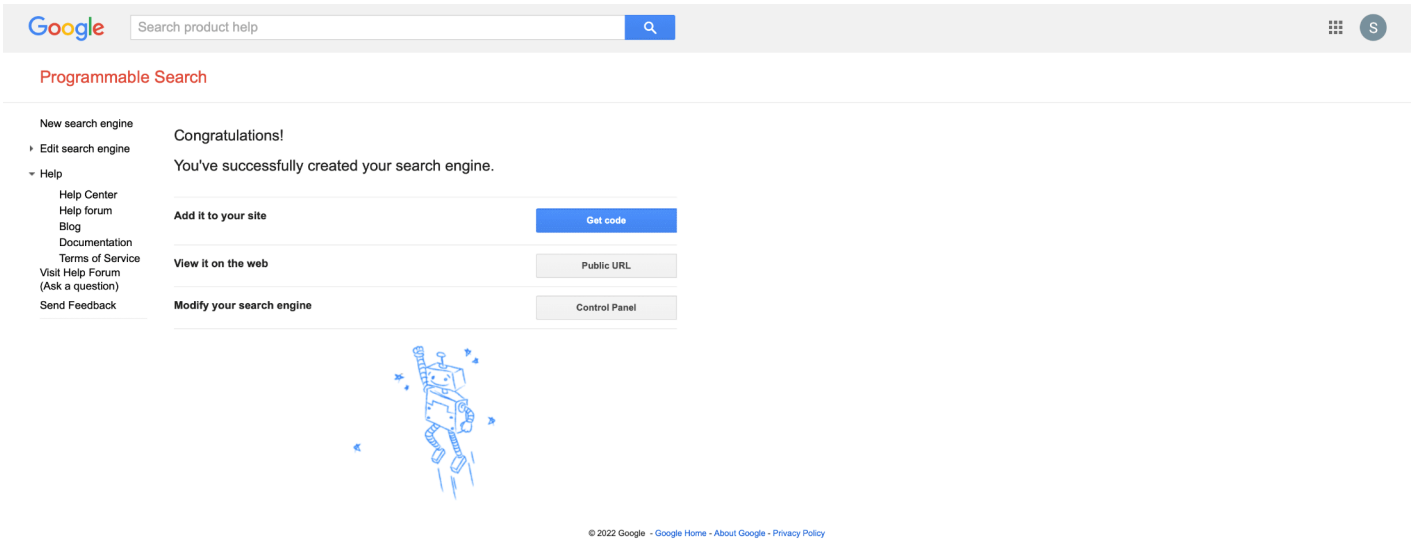
- masteringionic.com
- angular.io
- capacitorjs.com
- stenciljs.com

- ionicframework.com
- firebase.com

I have also given this Programmable Search Engine the name **Masteringionic**:

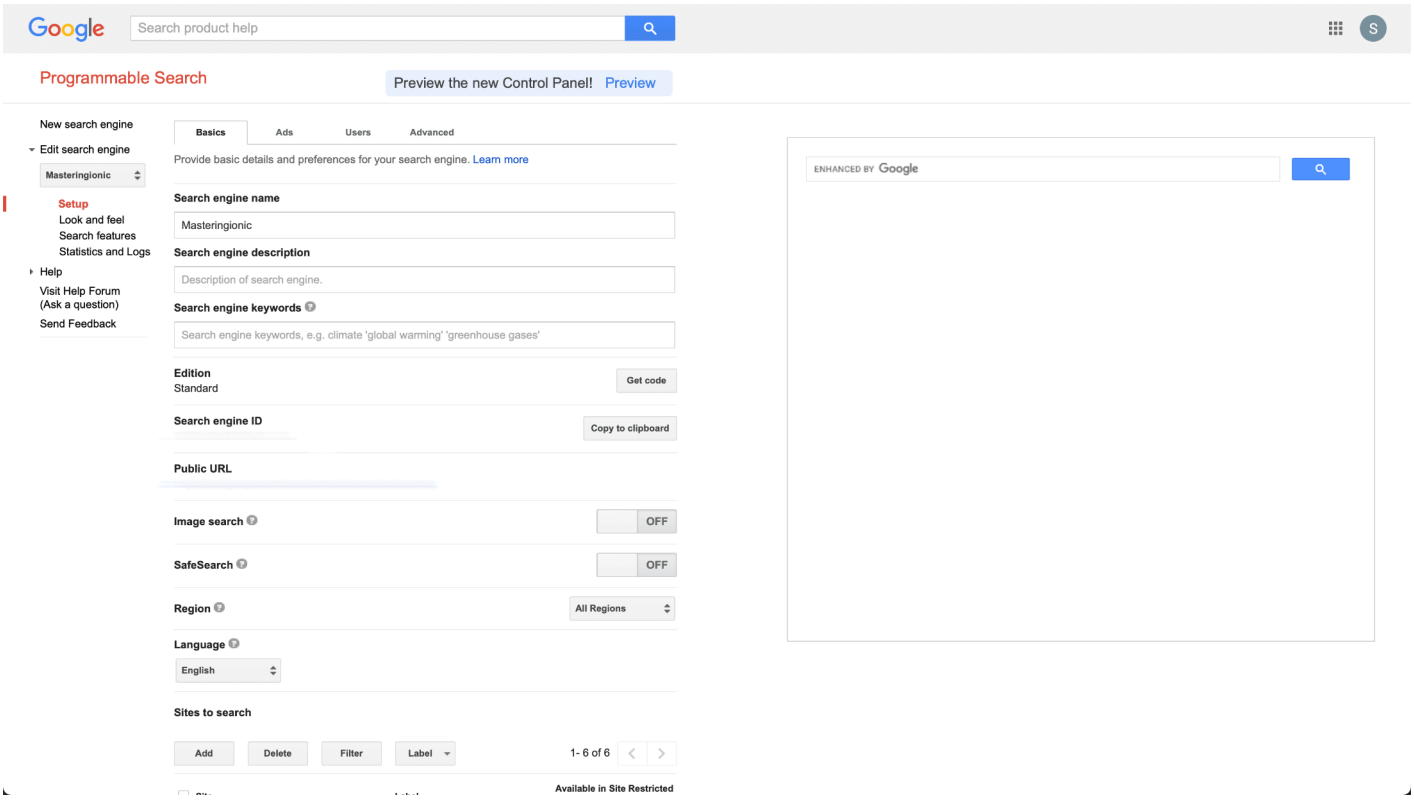


Once you have completed the reCaptcha and clicked on the Create button you will be greeted with the following congratulations screen:



Notice the **Edit search engine** link in the sub-menu on the left hand-side of the screen? Click on this as we need to access the **Search engine ID** (this is important as this forms an essential part of the query string that we need to supply when making HTTP calls to the Custom Search JSON API).

The **Search engine ID** is located under the **Basics** tab as shown below - copy this value and store this in a temporary text file for safekeeping (we'll come back to this later):



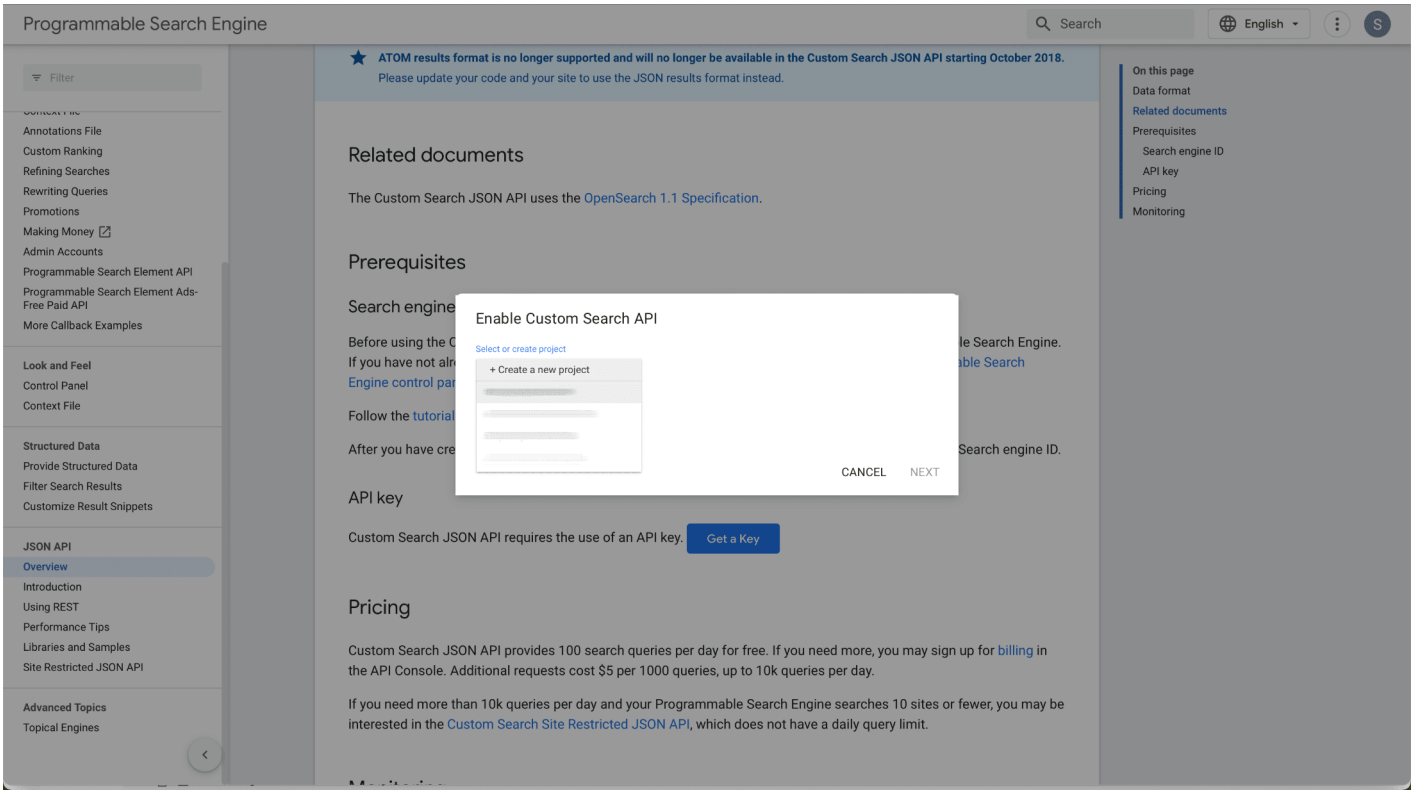
With our Programmable search engine now in place all that remains is to set up an API key to allow usage of the Custom Search JSON API within our project.

Fortunately this is a very simple and quick process - open your system browser and navigate to this URL: https://developers.google.com/custom-search/v1/overview.

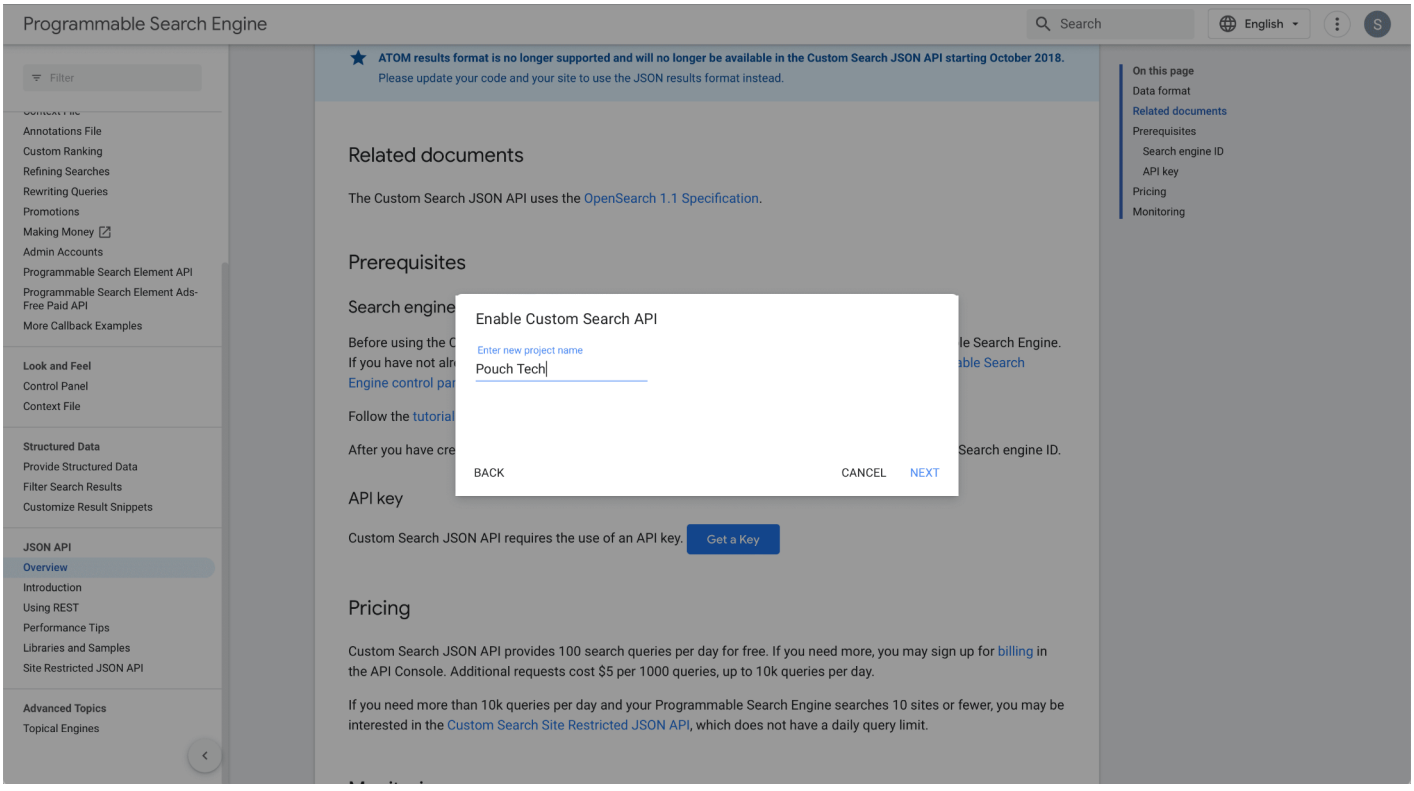
Under the **Prerequisites** section click onto the **Get a key** button in the sub-section titled **API key**.

If you are not already logged into your Google Developer account you will be redirected and prompted to do so - upon successful login you will then be redirected back to this page.

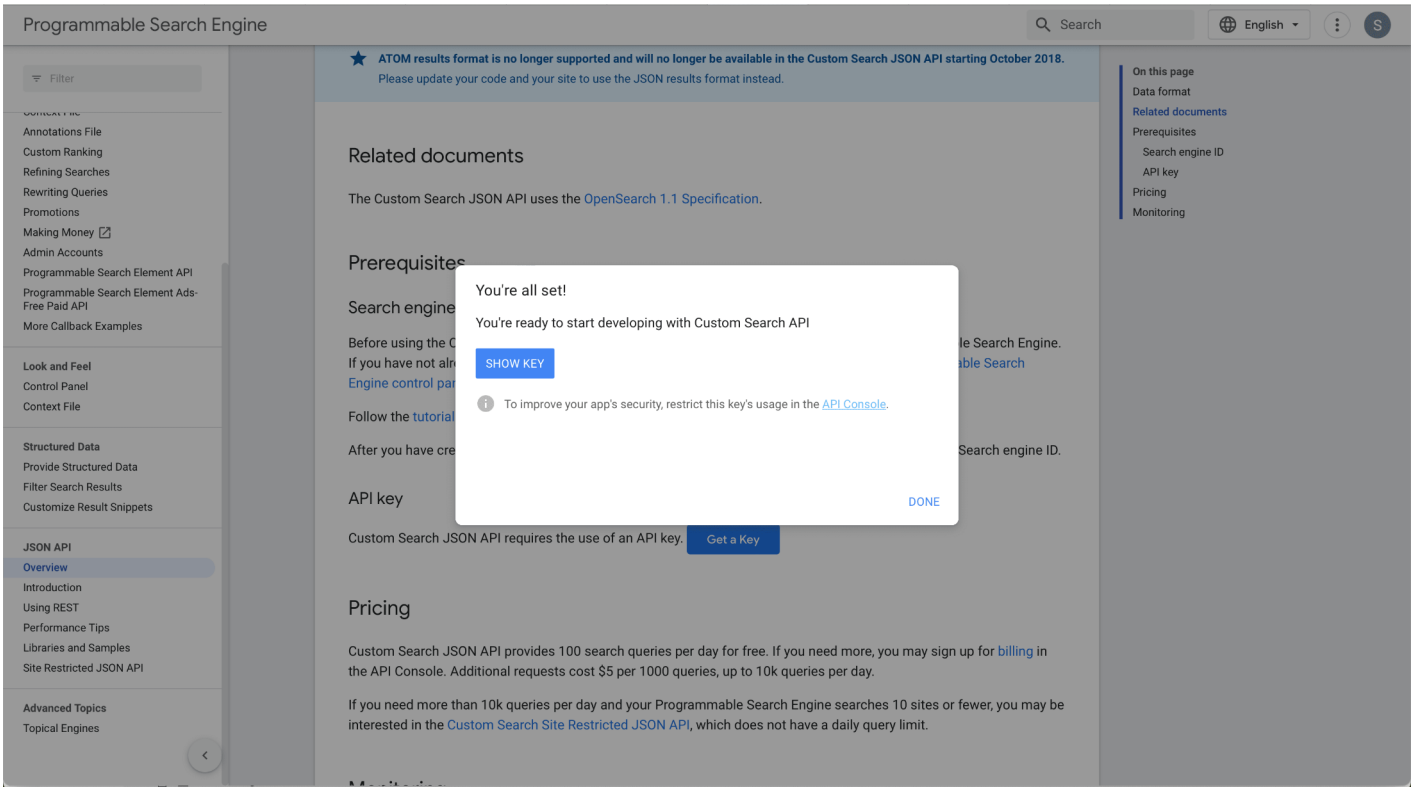
Once logged in (or if already logged in) you will be presented with a modal titled **Enable Custom Search API** that displays a dropdown menu listing your existing Google projects (if any) and an option to **Create a new project** like so:



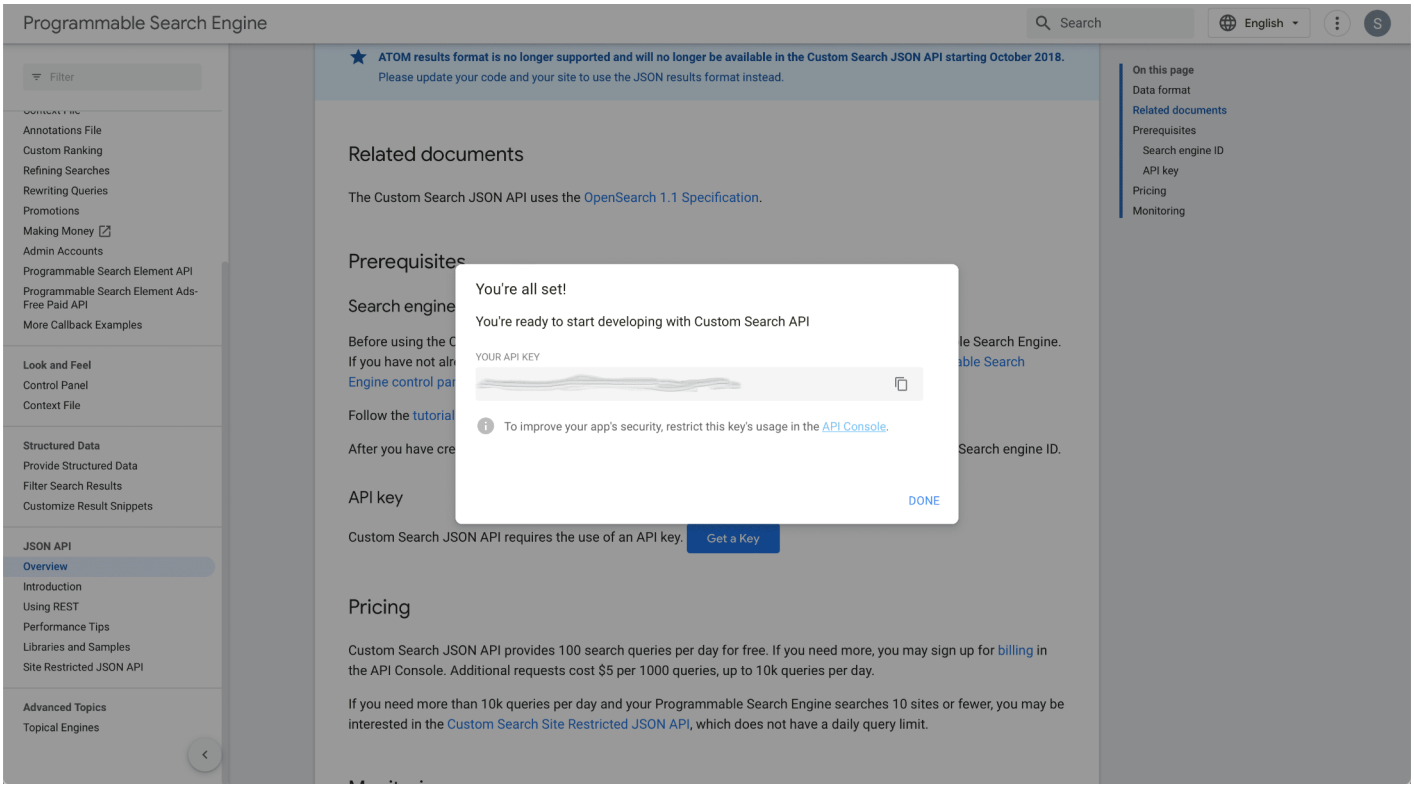
Enter a name for the new project (I.e. **Pouch Tech** below) and click next:



Generating the project may take a few seconds to complete so sit back and relax until you see the following **You're all set!** Message:

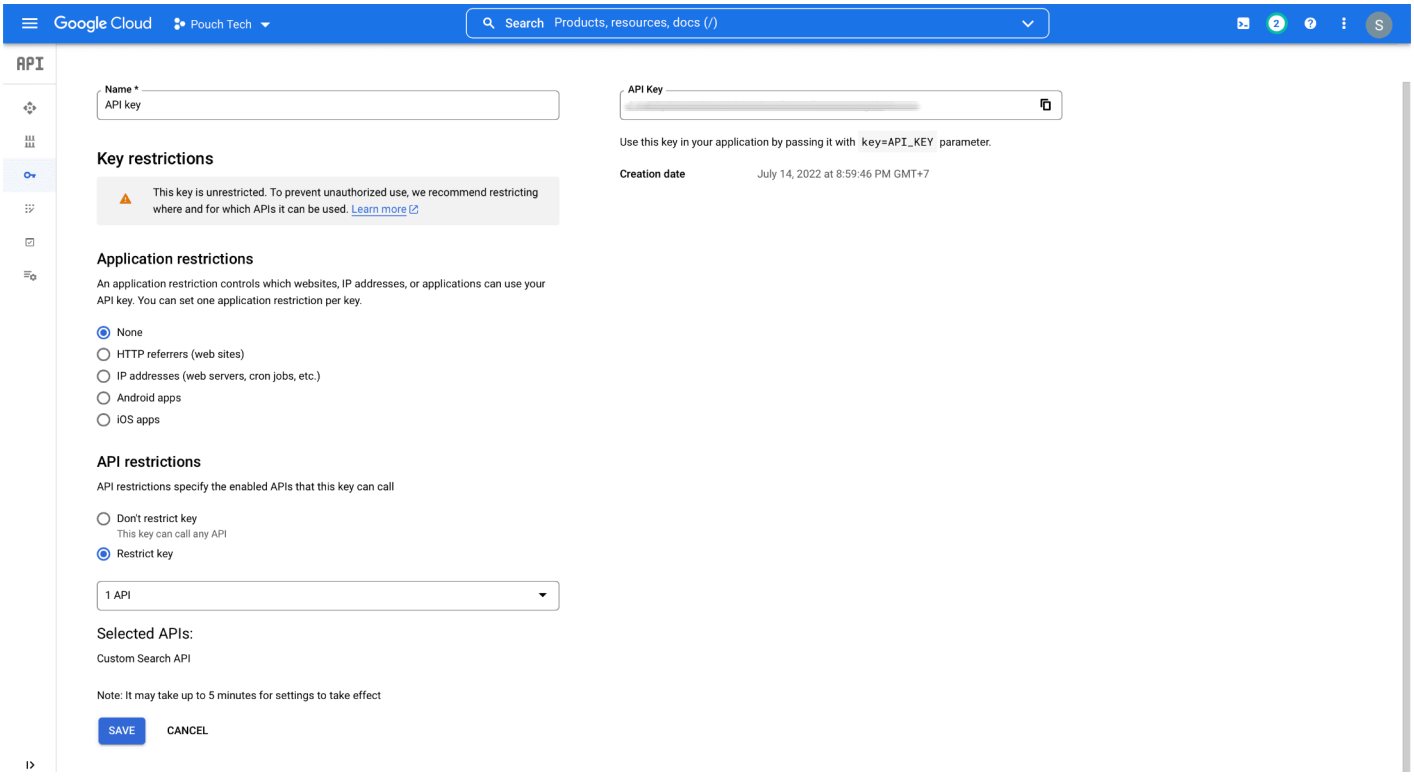


Click on the **Show Key** button to reveal the API key that is associated with the Programmable Search Engine (and that you'll need to use for making HTTP calls to the Custom Search JSON API):



Similar to the Search Engine ID value copy and paste the displayed API key value into a temporary text file for safekeeping (we'll use this shortly).

Next - restrict the API key usage by clicking on the API Console link in the modal.



This redirects to the Google Cloud project **Key restrictions** page.

Under the **API restrictions** section select the **Restrict key** radio button. This will then present a dropdown menu from which you need to select the **Custom Search API** option.

Once done click on the **Save** button and, with that, all necessities for our API configuration are now concluded!

With all of our backend configurations firmly in place let's now return to our Ionic project and begin building out the foundations that we generated earlier via the CLI tool - starting with our environment files.

Environment is everything

Remember those Search Engine ID and API key values that I asked you to copy and paste into a temporary text file for safekeeping?

Now's the time to integrate those into our Ionic project.

Within the **pouchdb-technologies/src/environments** directory open the following

files contained within there:

- **environment.ts**
- **environment.prod.ts**

Within each of these environment configuration files add the following block of code (substituting the placeholder texts with your copied values) inside the **environment** object as shown below (additions are highlighted in bold and I have chosen the **environment.ts** file as an example here but the same applies to both environment files):

```
export const environment = {
  production: false,
  settings: {
    keys: {
      searchEngineAPI: 'YOUR-GOOGLE-PROJECT-API-KEY-VALUE-HERE',
      searchEngineId: 'YOUR-SEARCH-ENGINE-ID-VALUE-HERE'
    },
    databases: {
      pouchdb: {
        name: 'technologies'
      }
    }
};
```

Here we simply define a **settings** object which declares property references for our Google project API key and search engine Id values and pouchDB database name.

The beauty of using our environment files to implement site wide configurations such as database values and API keys is that these are located in one central, intuitive and easy to manage location within the project instead of being scattered throughout different components or services.

We can always add further configurations to these environment files if we were to build out the project with additional APIs, third-party software integrations (which require their own configuration values to be implemented) and so on and so forth.

End of preview