The Modern Application Stack – MEAN, MERN, and More

October 2017

mongoDB

Table of Contents

Introducing the MEAN and MERN stacks

This ebook examines the technologies that are driving the development of modern web and mobile applications, notably the **MERN** and **MEAN** stacks. The book goes on to step through tutorials to build all layers of an application. The final chapter builds an application using MongoDB Stitch, the new Backend as a Service (BaaS) for applications built on MongoDB.

Users increasingly demand a far richer experience from web sites – expecting the same level of performance and interactivity they get with native desktop and mobile apps. At the same time, there's pressure on developers to deliver new applications faster and continually roll-out enhancements, while ensuring that the application is highly available and can be scaled appropriately when needed. Fortunately, there's a (sometimes bewildering) set of enabling technologies that make all of this possible.

If there's one thing that ties these technologies together, it's JavaScript and its successors (ES6, TypeScript, JSX, etc.) together with the JSON data format. The days when the role of JavaScript was limited to adding visual effects like flashing headers or pop-up windows are past. Developers now use JavaScript to implement the frontend experience as well as the application logic and even to access the database. There are two dominant JavaScript web app stacks – MEAN (**M**ongoDB, **E**xpress, **A**ngular, **N**ode.js) and MERN (**M**ongoDB, **E**xpress, **R**eact, **N**ode.js) and so we'll use those as paths to guide us through the ever expanding array of tools and frameworks.

This first chapter serves as a primer for many of these technologies. Subsequents chapters take a deep dive into specific topics – working through the end-to-end development of *Mongopop* - an application to populate a MongoDB database with realistic data and then perform other operations on that data.

The MEAN stack

We'll start with MEAN as it's the more established stack but most of what's covered here is applicable to **MERN** (swap Angular with React).

MEAN is a set of Open Source components that together, provide an end-to-end framework for building dynamic web applications; starting from top (code running in the

browser) to the bottom (database). The stack is made up of:

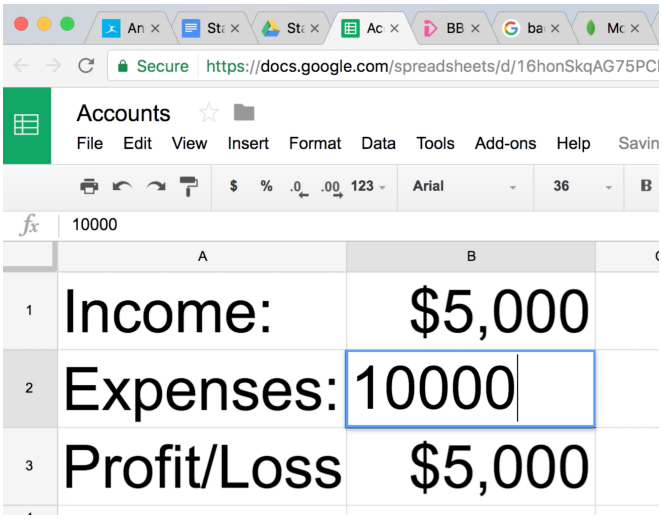
- **A**ngular (formerly Angular.js, now also known as Angular 2): Frontend web app framework; runs your JavaScript code in the users browser, allowing your application UI to be dynamic
- **E**xpress (sometimes referred to as Express.js): Backend web application framework running on top of Node.js
- **N**ode.js : JavaScript runtime environment – lets you implement your application backend in JavaScript
- **M**ongoDB : Document database – used by your backend application to store its data as JSON (JavaScript Object Notation) documents

A common theme in the MEAN stack is JavaScript – every line of code you write can be in the same language. You even access the database using MongoDB's native, *Idiomatic* JavaScript/Node.js driver. What do we mean by idiomatic? Using the driver feels natural to a JavaScript developer as all interaction is performed using familiar concepts such as JavaScript objects and asynchronous execution using either callback functions or promises (explained later). Here's an example of inserting an array of 3 JavaScript objects:

```
myCollection.insertMany([
  {name: {first: "Andrew", last: "Morgan"},
  {name: {first: "Elvis"}, died: 1977},
  {name: {last: "Mainwaring", title: "Captain"},
    born: 1885}
  ])
 .then(
  function(results) {
  resolve(results.insertedCount);
},
function(err) {
  console.log("Failed to insert Docs: " +
                  err.message);
  reject(err);
})
```

Angular 2

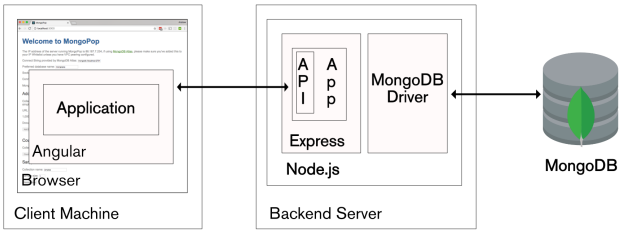
Angular, originally created and maintained by Google, runs your JavaScript code within the user's web browsers to implement a *reactive* user interface (UI). A reactive UI gives the user immediate feedback as they give their input (in contrast to static web forms where you enter all of your data, hit "Submit" and wait.



Version 1 of Angular was called AngularJS but it was shortened to Angular in Angular 2 after it was completely rewritten in Typescript (a superset of JavaScript) – Typescript is now also the recommended language for Angular apps to use.

You implement your application frontend as a set of *components* – each of which consists of your JavaScript (TypeScript) code and an HTML template that includes hooks to execute and use the results from your TypeScript functions. Complex application frontends can be crafted from many simple (optionally nested) components.

Angular application code can also be executed on the backend server rather than in a browser, or as a native desktop or mobile application.



Express

Express is the web application framework that runs your backend application (JavaScript) code. Express runs as a module within the Node.js environment.

Express can handle the routing of requests to the right parts of your application (or to different apps running in the same environment).

You can run the app's full business logic within Express and even generate the final HTML to be rendered by the user's browser. At the other extreme, Express can be used to simply provide a REST API – giving the frontend app access to the resources it needs e.g., the database.

In this book, we will use Express to perform two functions:

- Send the frontend application code to the remote browser when the user browses to our app
- Provide a REST API that the frontend can access using HTTP network calls, in order to access the database

Node.js

Node.js is a JavaScript runtime environment that runs your backend application (via Express).

Node.js is based on Google's V8 JavaScript engine which is used in the Chrome browsers. It also includes a number of modules that provides features essential for implementing web applications – including networking protocols such as HTTP. Third party modules, including the MongoDB driver, can be installed, using the `npm` tool.

Node.js is an asynchronous, event-driven engine where the application makes a request and then continues working on other useful tasks rather than stalling while it waits for a response. On completion of the requested task, the application is informed of the results via a callback. This enables large numbers of operations to be performed in parallel which is essential when scaling applications. MongoDB was also designed to be used asynchronously and so it works well with Node.js applications.

MongoDB

MongoDB is an open-source, document database that provides persistence for your application data and is designed with both scalability and developer agility in mind. MongoDB bridges the gap between key-value stores, which are fast and scalable, and relational databases, which have rich functionality. Instead of storing data in rows and columns as one would with a relational database, MongoDB stores JSON documents in collections with dynamic schemas.

MongoDB's document data model makes it easy for you to store and combine data of any structure, without giving up sophisticated validation rules, flexible data access, and rich indexing functionality. You can dynamically modify the schema without downtime – vital for rapidly evolving applications.

It can be scaled within and across geographically distributed data centers, providing high levels of availability and scalability. As your deployments grow, the database scales easily with no downtime, and without changing your application.

MongoDB Atlas is a database as a service for MongoDB, letting you focus on apps instead of ops. With MongoDB Atlas, you only pay for what you use with a convenient hourly billing model. With the click of a button, you can scale up and down when you need to, with no downtime, full security, and high performance.

Note that MongoDB also offers a Backend as a Service (BaaS), called MongoDB Stitch. The final chapter of this book will demonstrate how much of the work of implementing and interfacing with your application backend is greatly simplified when using Stitch.

Our application will access MongoDB via the JavaScript/Node.js driver which we install as a Node.js module.

What's done where?

tl;dr – it's flexible.

There is clear overlap between the features available in the technologies making up the MEAN stack and it's important to decide "who does what".

Perhaps the biggest decision is where the application's "hard work" will be performed. Both Express and Angular include features to route to pages, run application code, etc. and either can be used to implement the business logic for sophisticated applications. The more traditional approach would be to do it in the backend in Express. This has several advantages:

- Likely to be closer to the database and other resources and so can minimize latency if lots of database calls are made

- Sensitive data can be kept within this more secure environment
- Application code is hidden from the user, protecting your intellectual property
- Powerful servers can be used – increasing performance

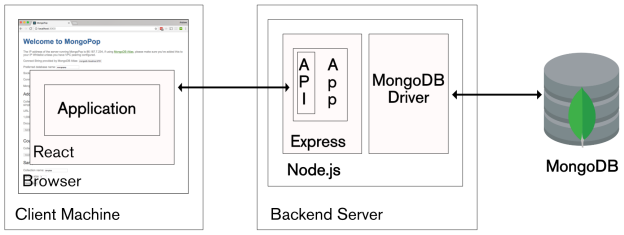
However, there's a growing trend to push more of the functionality to Angular running in the user's browser. Reasons for this can include:

- Use the processing power of your users' machines; reducing the need for expensive resources to power your backend. This provides a more scalable architecture, where every new user brings their own computing resources with them.
- Better response times (assuming that there aren't too many trips to the backend to access the database or other resources)
- *Progressive Applications*. Continue to provide (probably degraded) service when the client application cannot contact the backend (e.g. when the user has no internet connection). Modern browsers allow the application to store data locally and then sync with the backend when connectivity is restored.

Perhaps, a more surprising option for running part of the application logic is within the database. MongoDB has a sophisticated aggregation framework which can perform a lot of analytics – often more efficiently than in Express or Angular as all of the required data is local.

Another decision is where to validate any data that the user supplies. Ideally this would be as close to the user as possible – using Angular to check that a provided password meets security rules allows for instantaneous feedback to the user. That doesn't mean that there isn't value in validating data in the backend as well, and using MongoDB's document validation functionality can guard against buggy software writing erroneous data.

ReactJS – rise of the MERN stack



An alternative to Angular is React (sometimes referred to as ReactJS), a JavaScript library developed by Facebook to build interactive/reactive user interfaces. Like Angular, React breaks the frontend application down into components. Each component can hold its own state and a parent can pass its state down to its child components and those components can pass changes back to the parent through the use of callback functions.

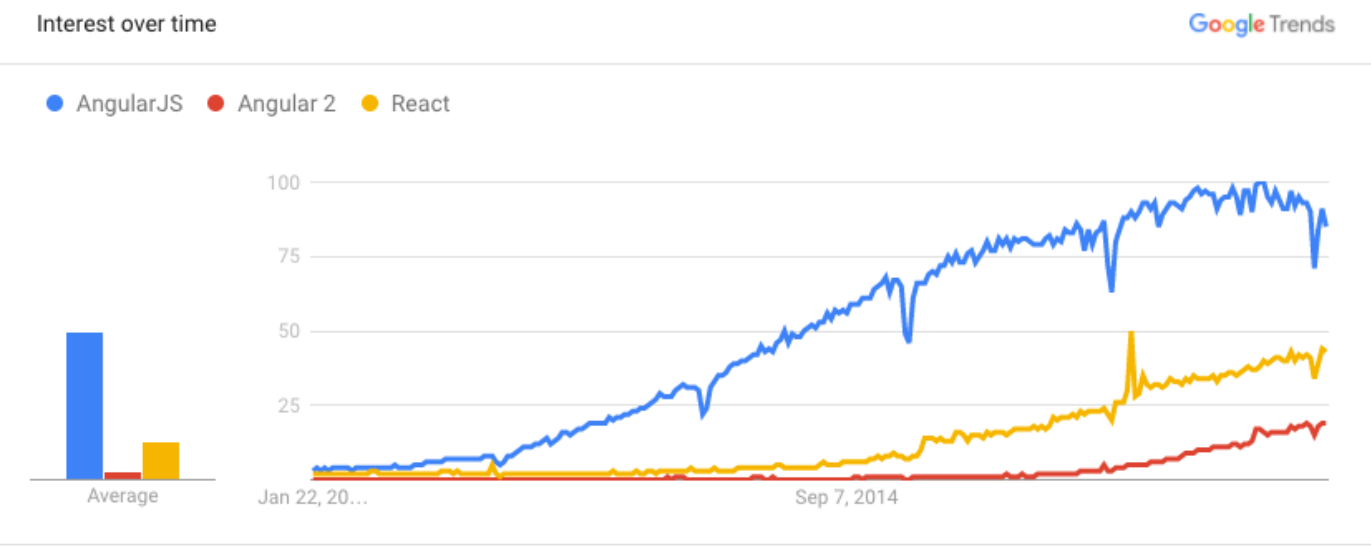
React components are typically implemented using JSX – an extension of JavaScript that allows HTML syntax to be embedded within the code:

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}
      </div>;
  }
}
```

React is most commonly executed within the browser but it can also be run on the backend server within Node.js, or as a mobile app using *React Native*.

So should you use Angular 2 or React for your new web application? A quick Google search will find you some fairly deep comparisons of the two technologies but in summary, Angular 2 is a little more powerful while React is easier for developers to get up to speed with and use. This book builds a near-identical web app using first the MEAN and then the MERN stack – hopefully helping you pick a favorite.

The following snapshot from Google Trends suggests that Angular has been much more common for a number of years but that React is gaining ground:



Worldwide. Past 5 years.

Why are these stacks important?

Having a standard application stack makes it much easier and faster to bring in new developers and get them up to speed as there's a good chance that they've used the technology elsewhere. For those new to these technologies, there exist some great resources to get you up and running.

From MongoDB upwards, these technologies share a common aim – look after the critical but repetitive stuff in order to free up developers to work where they can really add value: building your killer app in record time.

These are the technologies that are revolutionizing the web, building web-based services that look, feel, and perform just as well as native desktop or mobile applications.

The separation of layers, and especially the REST APIs, has led to the breaking down of application silos. Rather than an application being an isolated entity, it can now interact with multiple services through public APIs:

1. Register and log into the application using my Twitter account
2. Identify where I want to have dinner using Google Maps and Foursquare
3. Order an Uber to get me there
4. Have Hue turn my lights off and Nest turn my heating down
5. Check in on Facebook
6. ...

Variety & constant evolution

Even when constraining yourself to the JavaScript ecosystem, the ever-expanding array of frameworks, libraries, tools, and languages is both impressive and intimidating at the same time. The great thing is that if you're looking for some middleware to perform a particular role, then the chances are good that someone has already built it – the hardest part is often figuring out which of the 5 competing technologies is the best fit for you.

To further complicate matters, it's rare for the introduction of one technology not to drag in others for you to get up to speed on: Node.js brings in `npm`; Angular 2 brings in Typescript, which brings in `tsc`; React brings in ES6, which brings in Babel; ....

And of course, none of these technologies are standing still and new versions can require a lot of up-skilling to use – Angular 2 even moved to a different programming language!

The evolution of JavaScript

The JavaScript language itself hasn't been immune to change.

Ecma International was formed to standardize the language specification for JavaScript (and similar language forks) to increase portability – the ideal being that any "JavaScript" code can run in any browser or other JavaScript runtime environment.

The most recent, widely supported version is ECMAScript 6 – normally referred to as *ES6*. ES6 is supported by recent versions of Chrome, Opera, Safari, and Node.js). Some platforms (e.g. Firefox and Microsoft Edge) do not yet support all features of ES6. These are some of the key features added in ES6:

- Classes & modules
- Promises – a more convenient way to handle completion or failure of synchronous function calls (compared to callbacks)
- Arrow functions – a concise syntax for writing function expressions
- Generators – functions that can yield to allow others to execute
- Iterators
- Typed arrays

Typescript is a superset of ES6 (JavaScript); adding static type checking. Angular 2 is written in Typescript and Typescript is the primary language to be used when writing code to run in Angular 2.

Because ES6 and Typescript are not supported in all environments, it is common to *transpile* the code into an earlier version of JavaScript to make it more portable. In this book, `tsc` is used to transpile Typescript into JavaScript while the React example uses Babel (via `react-script`) to transpile our ES6 code.

And of course, JavaScript is augmented by numerous libraries. The Angular 2 chapter uses *Observables* from the RxJS reactive libraries which greatly simplify making asynchronous calls to the backend (a pattern historically referred to as AJAX).

Chapter summary

This chapter has introduced some of the technologies used to build modern, reactive, mobile and web applications – most notably the MEAN and MERN stacks. The upcoming chapters steps through building the MongoPop application:

- Using MongoDB With Node.js
- Building a REST API Using Express.js
- Building a Client UI Using Angular 2 (formerly AngularJS) & TypeScript
- Using ReactJS & ES6 to Build a UI (the rise of MERN)
- Browsers Aren't the Only UI

As already covered, the MERN and MEAN stacks are evolving rapidly and new JavaScript frameworks are being added all of the time. Inevitably, some of the details in this book will become dated but the concepts covered will remain relevant.

Using MongoDB With Node.js

This chapter demonstrates how to use MongoDB from Node.js.

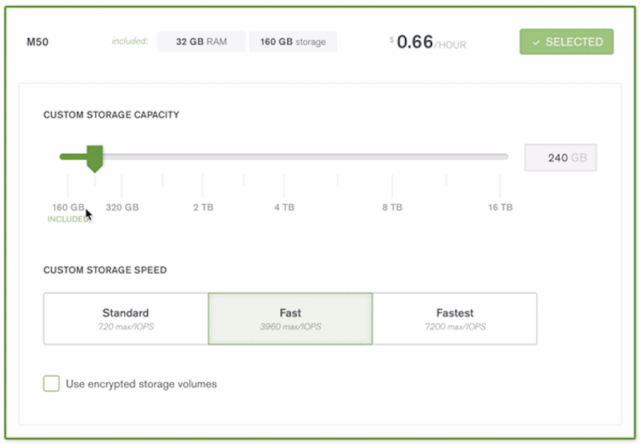
The application – Mongopop

MongoPop is a web application that can be used to help you test out and exercise MongoDB. After supplying it with the database connection information (e.g., as displayed in the MongoDB Atlas GUI), MongoPop provides these features:

- Accept your username and password and create the full MongoDB connect string – using it to connect to your database
- Populate your chosen MongoDB collection with bulk data (created with the help of the Mockeroo service)
- Count the number of documents
- Read sample documents
- Apply bulk changes to selected documents

Downloading, running, and using the Mongopop application

Rather than installing and running MongoDB ourselves, it's simpler to spin one up in MongoDB Atlas:



To get the application code, either download and extract the zip file or use `git` to clone the Mongopop repo:

```
git clone git@github.com:am-MongoDB/MongoDB-Mongopop.git
cd MongoDB-Mongopop
```

If you don't have Node.js installed then that needs to be done before building the application; it can be downloaded from nodejs.org .

A file called `package.json` is used to control `npm` (the package manager for Node.js); here is the final version for the application:

```
{
  "//": "This contains the meta data to run the
  "//": Mongopop application",

  "name": "Mongopop",
  "description": "Adds and manipulates data with
        MongoDB Atlas or other MongoDB databases",
  "author":
    "Andrew Morgan andrew.morgan@mongodb.com",
  "version": "0.0.1",
  "private": false,

  "//": "The scripts are used to build, launch and
  "//": debug both the server and",
  "//": "client side apps. e.g. `npm install`
  "//": will install the client and",
  "//": "server-side dependencies",
  "scripts": {
    "start": "cd public && npm run tsc && cd ..
        && node ./bin/www",
    "debug": "cd public && npm run tsc && cd ..
        && node-debug ./bin/www",
    "tsc": "cd public && npm run tsc",
    "tsc:w": "cd public && npm run tsc:w",
    "express": "node ./bin/www",
    "express-debug": "node-debug ./bin/www",
    "postinstall": "cd public && npm install"
  },

"//": "The Node.js modules that are needed for
  "//": the server-side application (the",
  "//": "client-side dependencies are specified
  "//": in public/package.jsom).",

"dependencies": {
    "body-parser": "~1.15.1",
    "debug": "~2.2.0",
    "ee-first": "^1.1.1",
    "ejs": "^2.5.2",
    "express": "^4.13.4",
    "external-ip": "^0.2.4",
    "jade": "~1.11.0",
    "methods": "^1.1.2",
    "mongodb": "^2.2.5",
    "morgan": "~1.7.0",
    "pug": "^0.1.0",
    "request": "^2.74.0",
    "serve-favicon": "~2.3.0",
    "tsc": "^1.20150623.0"
  }
}
```

The `scripts` section defines a set of shortcuts that can be executed using `npm run script-name`. For example `npm run debug` runs the Typescript transpiler (`tsc`) and then the Express framework in debug mode. `start` is a special case and can be executed with `npm start`.

Before running any of the software, the Node.js dependencies must be installed (into the `node_modules` directory):

```
npm install
```

Note the list of dependencies in `package.json` – these are the Node.js packages that will be installed by `npm`

`install`. After those modules have been installed, `npm` will invoke the `postinstall` script (that will be covered later). If you later realize that an extra package is needed then you can install it and add it to the dependency list with a single command. For example, if the MongoDB Node.js driver hadn't already been included then it could be added with `npm install --save mongodb` – this would install the package as well as saving the dependency in `package.json`.

The application can then be run:

```
npm start
```

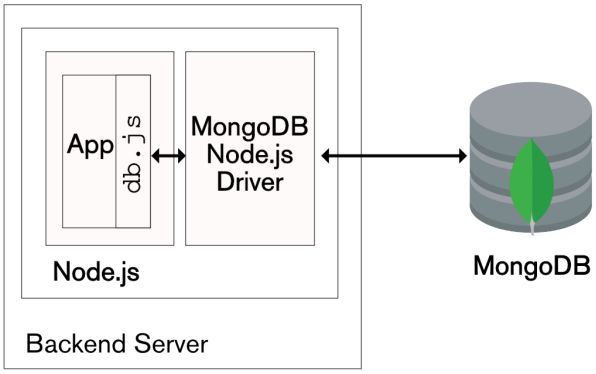
Once running, browse to `http://localhost:3000/` to try out the application. When browsing to that location, you should be rewarded with the IP address of the server where Node.js is running (useful when running the client application remotely) – this IP address must be added to the *IP Whitelist* in the Security tab of the MongoDB Atlas GUI. Fill in the password for the MongoDB user you created in MongoDB Atlas and you're ready to go. Note that you should get your own URL, for your own data set using the Mockaroo service – allowing you to customize the format and contents of the sample data (and avoid exceeding the Mockaroo quota limit for the example URL).

What are all of these files?

- **`package.json`**: Instructs the Node.js package manager (`npm`) what it needs to do; including which dependency packages should be installed
- **`node_modues`**: Directory where `npm` will install packages
- **`node_modues/mongodb`**: The MongoDB driver for Node.js
- **`node_modues/mongodb-core`**: Low-level MongoDB driver library; available for framework developers (application developers should avoid using it directly)
- **`javascripts/db.js`**: A JavaScript module we've created for use by our Node.js apps (in this book, it will be Express) to access MongoDB; this module in turn uses the MongoDB Node.js driver.

The rest of the files and directories can be ignored for now – they will be covered in later chapters.

Architecture



The MongoDB Node.js Driver provides a JavaScript API which implements the network protocol required to read and write from a local or remote MongoDB database. If using a replica set (and you should for production) then the driver also decides which MongoDB instance to send each request to. If using a sharded MongoDB cluster then the driver connects to the `mongos` query router, which in turn picks the correct shard(s) to direct each request to.

We implement a shallow wrapper for the driver (`javascripts/db.js`) which simplifies the database interface that the application code (coming in the next chapter) is exposed to.

Code highlights

`javascripts/db.js` defines an /object prototype/ (think *class* from other languages) named `DB` to provide access to MongoDB.

Its only dependency is the MongoDB Node.js driver:

```
var MongoClient = require('mongodb').MongoClient;
```

The prototype has a single property – `db` which stores the database connection; it's initialized to `null` in the constructor:

```
function DB() {
  this.db = null; // The MongoDB connection
}
```

The MongoDB driver is asynchronous (the function returns without waiting for the requested operation to complete); there are two different patterns for handling this:

1. The application passes a *callback* function as a parameter; the driver will invoke this callback function when the operation has run to completion (either on success or failure)
2. If the application does not pass a callback function then the driver function will return a *promise*

This application uses the promise-based approach. This is the general pattern when using promises:

```
var _this = this;

myLibrary.myAsyncFunction(myParameters)
.then(
  function(myResults) {
    // If the asynchronous operation eventually
    // *succeeds* then the first of the `then`
    // functions is invoked and this code block
    // will be executed at that time.
    // `myResults` is an arbitrary name and it
    // is set to the result data sent back
    // by `myAsyncFunction` when resolving
    //the promise

    _this.results = myReults;
  },
  function(myError) {
    // If the asynchronous operation eventually
    // *fails* then the second of the `then`
    // functions is invoked and this code block
    // will be executed at that time.
    // `myError ` is an arbitrary name and it is
    // set to the error data sent back
    // by `myAsyncFunction` when rejecting the
    // promise

      console.log("Hit a problem: " +
          myError.message);
  }
)
```

The methods of the `DB` object prototype we create are also asynchronous and also return promises (rather than accepting callback functions). This is the general pattern for returning and then subsequently satisfying promises:

```
return new Promise(function(resolve, reject) {
  // At this point, control has already been
  // passed back to the calling function
  // and so we can safely perform operations that
  // might take a little time (e.g.
  // a big database update) without the
  // application hanging.

  var result = doSomethingThatTakesTime();

  // If the operation eventually succeeds then we
  // can invoke `resolve` (the name is
  // arbitrary, it just has to patch the first
  // function argument when the promise was
  // created) and optionally pass back the
  // results of the operation

  if (result.everythingWentOK) {
      resolve(result.documentsReadFromDatabase);
  } else {

      // Call `reject` to fail the promise and
      // optionally provide error information

      reject("Something went wrong: " +
          result.error.message);
  }
})
```

`db.js` represents a thin wrapper on top of the MongoDB driver library and so (with the background on promises under our belt) the code should be intuitive. The basic interaction model from the application should be:

1. Connect to the database
2. Perform all of the required database actions for the current request
3. Disconnect from the database

Here is the method from `db.js` to open the database connection:

```
DB.prototype.connect = function(uri) {

  // Connect to the database specified by the
  // connect string / uri

  // Trick to cope with the fact that "this" will
  //refer to a different object once in the
  //promise's function.
  var _this = this;

  // This method returns a javascript promise
  // (rather than having the caller
  // supply a callback function).

  return new Promise(function(resolve, reject) {
    if (_this.db) {
      // Already connected
      resolve();
    } else {
      var __this = _this;

    // Many methods in the MongoDB driver will
    // return a promise if the caller doesn't
    // pass a callback function.
    MongoClient.connect(uri)
    .then(
      function(database) {

        // The first function provided as a
        // parameter to "then"
        // is called if the promise is
        // resolved successfully. The
        // "connect" method returns the new
        // database connection
        // which the code in this function
        // sees as the "database"
        // parameter

        __this.db = database;

        // Indicate to the caller that the
        // request was completed successfully,
        // No parameters are passed back.

        resolve();
      },
      function(err) {

        // The second function provided as a
        // parameter to "then" is called if
        // the promise is rejected. "err" is
        // set to to the error passed by the
        // "connect" method.

        console.log("Error connecting: " +
          err.message);

        // Indicate to the caller that the
        // request failed and pass back
        // the error that was returned from
        // "connect"

        reject(err.message);
      }
      )
    }
  })
}
```

One of the simplest methods that can be called to use this new connection is to count the number of documents in a collection:

```
DB.prototype.countDocuments = function(coll) {

  // Returns a promise which resolves to the
  // number of documents in the
  // specified collection.

  var _this = this;

  return new Promise(function (resolve, reject){

    // {strict:true} means that the count operation
    // will fail if the collection doesn't yet
    // exist

    _this.db.collection(coll, {strict:true},
        function(error, collection){
      if (error) {
        console.log("Could not access collection: "
            + error.message);
        reject(error.message);
      } else {
          collection.count()
          .then(
            function(count) {
              // Resolve the promise with the count
              resolve(count);
            },
            function(err) {
              console.log("countDocuments failed: "
                  + err.message);
              // Reject the promise with the error
              // passed back by the count
              // function
              reject(err.message);
            }
          )
        }
    });
  })
}
```

Note that the `collection` method on the database connection doesn't support promises and so a callback function is provided instead.

And after counting the documents; the application should close the connection with this method:

```
DB.prototype.close = function() {

  // Close the database connection. This if the
  // connection isn't open then just ignore, if
  // closing a connection fails then log the fact
  // but then move on. This method returns nothing
  // - the caller can fire and forget.

  if (this.db) {
    this.db.close()
    .then(
      function() {},
      function(error) {
        console.log(
          "Failed to close the database: " +
              error.message)
      }
    )
  }
}
```

Note that `then` also returns a promise (which is, in turn, resolved or rejected). The returned promise could be created in one of 4 ways:

1. The function explicitly creates and returns a new promise (which will eventually be resolved or rejected).
2. The function returns another function call which, in turn, returns a promise (which will eventually be resolved or rejected).
3. The function returns a value – which is automatically turned into a resolved promise.
4. The function throws an error – which is automatically turned into a rejected promise.

In this way, promises can be chained to perform a sequence of events (where each step waits on the resolution of the promise from the previous one). Using those 3 methods from `db.js`, it's now possible to implement a very simple application function:

```
var DB = require('./javascripts/db');

function count (MongoDBURI, collectionName) {

  var database = new DB;

  database.connect(MongoDBURI)
  .then(
    function() {
      // Successfully connected to the database
      // Make the database call and pass the
      // returned promise to the next stage
      return
          database.countDocuments(collectionName);
    },
    function(err) {
      // DB connection failed, add context to the
      // error and throw it (it will be
      // converted to a rejected promise
      throw("Failed to connect to the database: " +
          err);
    })
  // The following `.then` clause uses the promise
  // returned by the previous one.
  .then(
    function(count) {
      // Successfully counted the documents
      console.log(count + " documents");
      database.close();
    },
    function(err) {
      // Could have got here by either
      // `database.connect` or
      // `database.countDocuments` failing
      console.log("Failed to count the documents: "
          + err);
      database.close();
    })
}

count("mongodb://localhost:27017/mongopop",
    "simples");
```

That function isn't part of the final application – the actual code will be covered in the next chapter – but jump ahead and look at `routes/pop.js` if you're curious).

It's worth looking at the `sampleCollection` prototype method as it uses a database cursor . This method fetches a "random" selection of documents – useful when you want to understand the typical format of the collection's documents:

```
DB.prototype.sampleCollection = function(coll,
    numberDocs) {

    // Returns a promise which is either resolved
    // with an array of "numberDocs" from the
    // "coll" collection or is rejected with the
    // error passed back from the database driver.

    var _this = this;

    return new Promise(function (resolve, reject){
      _this.db.collection(coll, {strict:true},
          function(error, collection){
        if (error) {
          console.log("Could not access collection:
              " + error.message);
          reject(error.message);
        } else {

          // Create a cursor from the aggregation
          // request

          var cursor = collection.aggregate([
            {
              $sample: {size: parseInt(numberDocs)}
            }],
            { cursor: { batchSize: 10 } }
          )

          // Iterate over the cursor to access each
          // document in the sample result set.
          // Could use cursor.each() if we wanted
          // to work with individual documents here

          cursor.toArray(function(error, docArray)
          {
            if (error) {
              console.log(
                  "Error reading fron cursor: " +
                      error.message);
              reject(error.message);
            } else {
              resolve(docArray);
            }
          })
        }
    })
  })
}
```

Note that `collection.aggregate` doesn't actually access the database – that's why it's a synchronous call (no need for a promise or a callback) – instead, it returns a `cursor`. The cursor is then used to read the data from MongoDB by invoking its `toArray` method. As `toArray` reads from the database, it can take some time and so it is an asynchronous call, and a callback function must be provided (`toArray` doesn't support promises).

The rest of these database methods can be viewed in `db.js` but they follow a similar pattern. The Node.js MongoDB Driver API documentation explains each of the methods and their parameters.

Chapter summary

This stepped through how to install and use Node.js and the MongoDB Node.js driver. This is our first step in building a modern, reactive application using the MEAN and MERN stacks.

The chapter went on to describe the implementation of a thin layer that's been created to sit between the application code and the MongoDB driver. The layer is there to provide a simpler, more limited API to make application development easier. In other applications, the layer could add extra value such as making semantic data checks.

The next chapter adds the Express framework and uses it to implement a REST API to allow clients to send requests of the MongoDB database. That REST API will subsequently be used by the client application (using Angular, and then React).

Building a REST API using Express.js

Introduction

This chapter uses Express to build a REST API so that a remote client can work with MongoDB.

The REST API

A Representational State Transfer (REST) interface provides a set of operations that can be invoked by a remote client (which could be another service) over a network, using the HTTP protocol. The client will typically provide parameters such as a string to search for or the name of a resource to be deleted.

Many services provide a REST API so that clients (their own and those of 3rd parties) and other services can use the service in a well defined, loosely coupled manner. As an

example, the Google Places API can be used to search for information about a specific location:

```
curl "https://maps.googleapis.com/maps/api/place/\
  details/json?placeid=ChIJKxSwWSZgAUgR0tWM0zAkZBc\
  &key=AIzaSyC53qhhXAmPOsxc34WManoorp7SVN_Qezo"

{
 "html_attributions" : [],
 "result" : {
    "address_components" : [
     {
      "long_name" : "19",
      "short_name" : "19",
      "types" : [ "street_number" ]
     },
     {
      "long_name" : "Route des Allassins",
      "short_name" : "Route des Allassins",
      "types" : [ "route" ]
     },
     {
      "long_name" : "Le Grand-Village-Plage",
      "short_name" : "Le Grand-Village-Plage",
      "types" : [ "locality", "political" ]
     },
     {
      "long_name" : "Charente-Maritime",
      "short_name" : "Charente-Maritime",
      "types" : [ "administrative_area_level_2",
          "political" ]
     },
     {
      "long_name" : "Nouvelle-Aquitaine",
      "short_name" : "Nouvelle-Aquitaine",
      "types" : [ "administrative_area_level_1",
          "political" ]
     },
     {
      "long_name" : "France",
      "short_name" : "FR",
      "types" : [ "country", "political" ]
     },
     {
      "long_name" : "17370",
      "short_name" : "17370",
        "types" : [ "postal_code" ]
     }
    ],
...

      "utc_offset" : 60,
    "vicinity" : "19 Route des Allassins,
        Le Grand-Village-Plage",
    "website" : "http://www.oleronvilla.com/"
 },
 "status" : "OK"
}
```

Breaking down the URI used in that `curl` request:

- No *method* is specified and so the `curl` command defaults to a HTTP `GET`.
- `maps.googleapis.com` is the address of the Google APIs service.
- `/maps/api/place/details/json` is the /route path/ to the specific operation that's being requested.
- `placeid=ChIJKxSwWSZgAUgR0tWM0zAkZBc` is a *parameter* (passed to the function bound to this route path), identifying which place we want to read the data for.
- `key=AIzaSyC53qhhXAmPOsxc34WManoorp7SVN_Qezo` is a parameter indicating the Google API key, verifying that it's a registered application making the request (Google will also cap, or bill for, the number of requests made using this key).

There's a convention as to which HTTP method should be used for which types of operation:

- **GET**: Fetches data
- **POST**: Adds new data
- **PUT**: Updates data
- **DELETE**: Removes data

Mongopop's REST API breaks this convention and uses `POST` for some read requests (as it's simpler passing arguments than with `GET`).

These are the REST operations that will be implemented in Express for Mongopop:

Express

Express is the web application framework that runs your backend application (JavaScript) code. Express runs as a module within the Node.js environment.

Express can handle the routing of requests to the right functions within your application (or to different apps running in the same environment).

You can run the app's full business logic within Express and even use an optional *view engine* to generate the final HTML to be rendered by the user's browser. At the other extreme, Express can be used to simply provide a REST API – giving the frontend app access to the resources it needs e.g., the database.

The Mongopop application uses Express to perform two functions:

Route Path	HTTP Method	Parameters	Response	Purpose
`/pop/`	GET		`{ "AppName": "MongoPop", "Version": 1.0 }`	Returns the version of the API.
`/pop/ip`	GET		`{"ip": string}`	Fetches the IP Address of the server running the Mongopop backend.
`/pop/config`	GET		`{ mongodb: { defaultDatabase: string, defaultCollection: string, defaultUri: string }, mockarooUrl: string }`	Fetches client-side defaults from the back-end config file.
`/pop/addDocs`	POST	`{ MongoDBURI: string, collectionName: string, dataSource: string, numberDocs: number, unique: boolean }`	`{ success: boolean, count: number, error: string }`	Add `numberDocs` batches of documents, using documents fetched from `dataSource`
`/pop/sampleDocs`	POST	`{ MongoDBURI: string, collectionName: string, numberDocs: number }`	`{ success: boolean, documents: string, error: string }`	Read a sample of the documents from a collection.
`/pop/countDocs`	POST	`{ MongoDBURI: string, collectionName: string }`	`{ success: boolean, count: number, error: string }`	Counts the number of documents in the collection.
`/pop/updateDocs`	POST	`{ MongoDBURI: string, collectionName: string, matchPattern: Object, dataChange: Object, threads: number }`	`{ success: boolean, count: number, error: string }`	Apply an update to all documents in a collection which match a given pattern

- Send the frontend application code to the remote client when the user browses to our app
- Provide a REST API that the frontend can access using HTTP network calls, in order to access the database

Downloading, running, and using the application

The application's Express code is included as part of the Mongopop package you installed earlier.

What are all of these files?

A reminder of the files described earlier:

- **`package.json`**: Instructs the Node.js package manager (`npm`) on what it needs to do; including which dependency packages should be installed
- **`node_modues`**: Directory where `npm` will install packages
- **`node_modues/mongodb`**: The MongoDB driver for Node.js
- **`node_modues/mongodb-core`**: Low-level MongoDB driver library; available for framework developers (application developers should avoid using it directly)
- **`javascripts/db.js`**: A JavaScript module we've created for use by our Node.js apps (in this book, it will be Express) to access MongoDB; this module in turn uses the MongoDB Node.js driver.

Other files and directories that are relevant to our Express application:

- **`config.js`**: Contains the application–specific configuration options
- **`bin/www`**: The script that starts an Express application; this is invoked by the `npm start` script within the `package.json` file. Starts the HTTP server, pointing it to the `app` module in `app.js`
- **`app.js`**: Defines the main application module (`app`). Configures:
  - That the application will be run by Express
  - Which routes there will be & where they are located in the file system (`routes` directory)
  - What view engine to use (Jade in this case)
  - Where to find the /views/ to be used by the view engine (`views` directory)
  - What middleware to use (e.g. to parse the JSON received in requests)
  - Where the static files (which can be read by the remote client) are located (`public` directory)
  - Error handler for queries sent to an undefined route
- **`views`**: Directory containing the templates that will be used by the Jade view engine to create the HTML for any pages generated by the Express application (for this application, this is just the error page that's used in cases such as mistyped routes ("404 Page not found"))
- **`routes`**: Directory containing one JavaScript file for each Express route
  - **`routes/pop.js`**: Contains the Express application for the `/pop` route; this is the implementation of the Mongopop REST API. This defines methods for all of the supported route paths.
- **`public`**: Contains all of the static files that must be accessible by a remote client (e.g., our Angular to React apps). This is not used for the REST API and so can be ignored for now.

The rest of the files and directories can be ignored for now – they will be covered in later chapters.

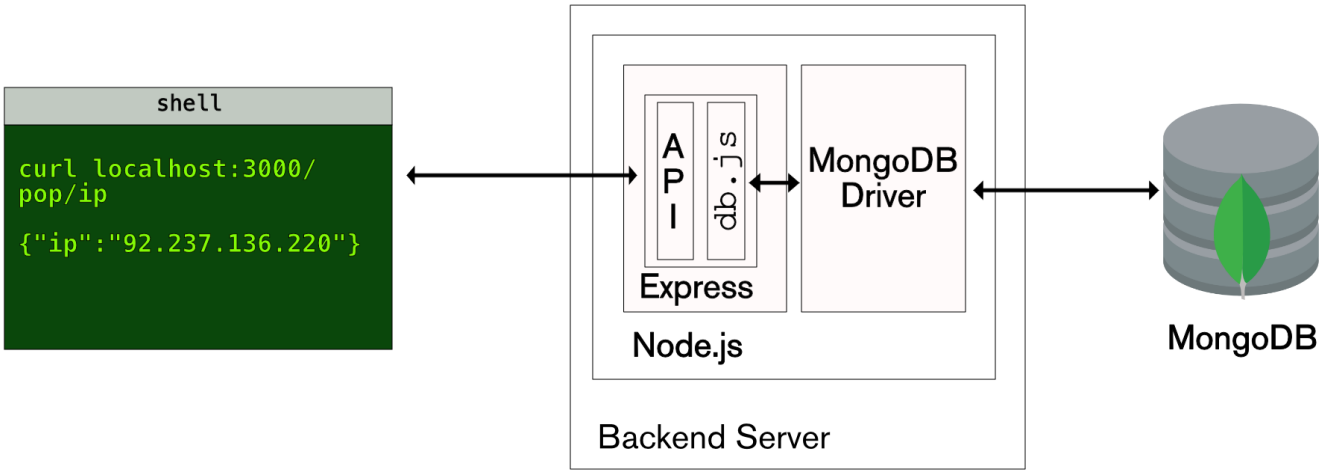
Architecture

The new REST API (implemented in `routes/pop.js`) uses the `javascripts/db.js` database layer implemented earlier to access the MongoDB database via the MongoDB Node.js Driver. As we don't yet have either the Angular or React clients, we will user the `curl` command-line tool to manually test the REST API.

Code highlights

config.js

```
var config = {
  expressPort: 3000,
  client: {
    mongodb: {
      defaultDatabase: "mongopop",
      defaultCollection: "simples",
      defaultUri: "mongodb://localhost:27017"
    },
    mockarooUrl: "http://www.mockaroo.com/\
        536ecbc0/download?count=1000&key=48da1ee0"
  }
};

module.exports = config;
```



The `config` module can be imported by other parts of the application so that your preferences can be taken into account.

`expressPort` is used by `bin/www` to decide what port the web server should listen on; change this if that port is already in use.

`client` contains defaults to be used by the client (Angular or React). It's important to create your own schema at Mockaroo.com and replace `client.mockarooUrl` with your custom URL (the one included here will fail if used too often).

`bin/www`

This is mostly boiler-plate code to start Express with your application. This code ensures that it is our application, `app.js`, that is run by the Express server:

```
// Location of the main express application module
// (`app.js`)
var app = require('../app');

var server = http.createServer(app);
```

This code uses the `expressPort` from `config.js` as the port for the server to listen on; it will be overruled if the user sets the `PORT` environment variable:

```
var config = require('../config.js');

var port = normalizePort(process.env.PORT ||
    config.expressPort);
app.set('port', port);
```

`app.js`

This file defines the `app` module ; much of the contents are boilerplate (and covered by comments in the code) but we look here at a few of the lines that are particular to this application.

Make this an Express application:

```
var express = require('express');
var app = express();
```

Define where the views (templates used by the Jade view engine to generate the HTML code) and static files (files that must be accessible by a remote client) are located:

```
app.set('views', path.join(__dirname, 'views'));
app.use(express.static(path.join(__dirname,
    'public')));
```

Create the `/pop` route and associate it with the file containing its code (`routes/pop.js`):

```
var pop = require('./routes/pop');
app.use('/pop', pop);
```

`routes/pop.js`

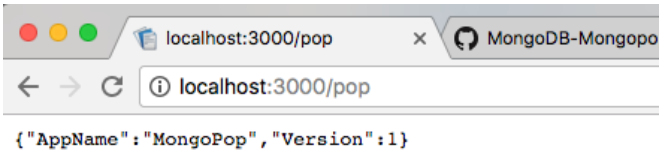
This file implements each of the operations provided by the Mongopop REST API. Because of the the `/pop` route defined in `app.js` Express will direct any URL of the form `http://mongopop-server:3000/pop/X` here. Within this file a route handler is created in order direct incoming requests to

`http://mongopop-server:3000/pop/X` to the appropriate function:

```
var router = express.Router();
```

As the `/pop` route is only intended for the REST API, end users shouldn't be browsing here but we create a top level handler for the `GET` method in case they do:

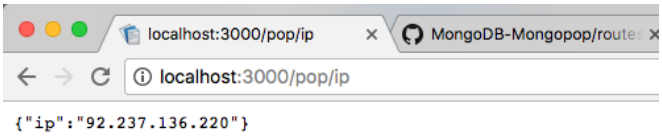
```
router.get('/', function(req, res, next) {

  // This isn't part of API and is just used from
  // a browser or curl to test that "/pop" is
  // being routed correctly.

var testObject = {
    "AppName": "MongoPop",
    "Version": 1.0
  }
  res.json(testObject);
});
```



This is the first time that we see how to send a response to a request; `res.json(testObject);` converts `testObject` into a JSON document and sends it back to the requesting client as part of the response message.

The simplest useful route path is for the `GET` method on `/pop/ip` which sends a response containing the IP address of the backend server. This is useful to the Mongopop client as it means the user can see it and add it to the MongoDB Atlas whitelist. The code to determine and store `publicIP` is left out here but can be found in the full source file for `pop.js`.

```
router.get('/ip', function(req, res, next) {

  // Sends a response with the IP address of the
  // server running this service.

  res.json({"ip": publicIP});
});
```



We've seen that it's possible to test `GET` methods from a browser's address bar; that isn't possible for `POST` methods and so it's useful to be able to test using the `curl` command-line command:

```
curl localhost:3000/pop/ip
{"ip":"92.237.136.220"}
```

The `GET` method for `/pop/config` is just as simple – responding with the client-specific configuration data:

```
var config = require('../config.js');

router.get('/config', function(req, res, next) {
    res.json(config.client);
})
```

```
curl http://localhost:3000/pop/config
{"mongodb":{"defaultDatabase":"mongopop",
  "defaultCollection":"simples",
  "defaultUri":"mongodb://localhost:27017"},
  "mockarooUrl":"http://www.mockaroo.com/536ecbc0\
  /download?count=1000&key=48da1ee0"}
```

The results of the request are still very simple but the output from `curl` is already starting to get messy; piping it through `python -mjson.tool` makes it easier to read:

```
curl http://localhost:3000/pop/config |
    python -mjson.tool
  % Total    % Received % Xferd  Average Speed ...
                                 Dload  Upload ...
100   195  100   195    0     0  28727      0  ...
{
  "mockarooUrl": "http://www.mockaroo.com/536ecbc0\
      /download?count=1000&key=48da1ee0",
  "mongodb": {
    "defaultCollection": "simples",
    "defaultDatabase": "mongopop",
    "defaultUri": "mongodb://localhost:27017"
  }
}
```

The simplest operation that actually accesses the database is the `POST` method for the `/pop/countDocs` route path:

```
var DB = require('../javascripts/db');

router.post('/countDocs', function(req, res, next)
  {

  /* Request from client to count the number of
  documents in a  collection; the request should be
  of the form:

  {
    MongoDBURI: string; // Connect string for
                        // MongoDB instance
    collectionName: string;
  }

  The response will contain:

  {
    success: boolean;
    count: number;      // The number of documents
                        // in the collection
    error: string;
  }
  */

  var requestBody = req.body;
  var database = new DB;

  database.connect(requestBody.MongoDBURI)
  .then(
    function() {
        return database.countDocuments(
            requestBody.collectionName)
    })
  .then(
    function(count) {
      return {
            "success": true,
            "count": count,
            "error": ""
      };
    },
    function(err) {
      console.log("Failed to count the documents: "
          + err);
      return {
        "success": false,
        "count": 0,
        "error": "Failed to count the documents: "
            + err
        };
    })
  .then(
    function(resultObject) {
      database.close();
      res.json(resultObject);
    })
})
```

`database` is an instance of the object prototype defined in `javascripts/db` and so all this method needs to do is use that object to:

- Connect to the database (using the address of the MongoDB server provided in the request body). The results from the promise returned by `database.connect` is passed to the function(s) in the first `.then` clause.
- The function in the `.then` clause handles the case where the `database.connect` promise is resolved (success). This function requests a count of the documents – the database connection information is now stored within the `database` object and so only the collection name needs to be passed. The promise returned by `database.countDocuments` is passed to the next `.then` clause. Note that there is no second (error) function provided, and so if the promise from `database.connect` is rejected, then that failure passes through to the next `.then` clause in the chain.
- The second `.then` clause has two functions:
  - The first is invoked if and when the promise is resolved (success) and it returns a success response (which is automatically converted into a resolved promise that it passed to the final `.then` clause in the chain). `count` is the value returned when the promise from the call to `database.countDocuments` was resolved.
  - The second function handles the failure case (could be from either `database.connect` or `database.countDocuments`) by returning an error response object (which is converted to a resolved promise).
- The final `.then` clause closes the database connection and then sends the HTTP response back to the client; the response is built by converting the `resultObject` (which could represent success or failure) to a JSON string.

Once more, `curl` can be used from the command-line to test this operation; as this is a `POST` request, the `--data` option is used to pass the JSON document to be included in the request:

```
curl -g -X POST --data '{
  "MongoDBURI": "mongodb://localhost:27017/\
      mongopop?authSource=admin&socketTimeoutMS=\
      30000&maxPoolSize=20",
  "collectionName": "simples"}' -i
  "localhost:3000/pop/countDocs"
  --header "Content-Type:application/json"

HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 43
ETag: W/"2b-+oXLqLmOvVK01tCeNbBDyg"
Date: Mon, 30 Jan 2017 08:35:16 GMT
Connection: keep-alive

{"success":true,"count":2828000,"error":""}
```

`curl` can also be used to test the error paths. Cause the database connection to fail by using the wrong port number in the MongoDB URI:

```
curl -g -X POST --data '{
  "MongoDBURI": "mongodb://localhost:27018/\
      mongopop?authSource=admin&socketTimeoutMS=\
      30000&maxPoolSize=20",
  "collectionName": "simples"}' -i
  "localhost:3000/pop/countDocs"
  --header "Content-Type:application/json"

HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 131
ETag: W/"83-h7jQCGc5BTG8dXp61IP4yA"
Date: Mon, 30 Jan 2017 08:37:44 GMT
Connection: keep-alive

{"success":false,"count":0,"error":"Failed to count\
 the documents: failed to connect to server\
 [localhost:27018] on first connect"}
```

Cause the count to fail by using the name of a non-existent collection:

```
curl -g -X POST --data '{
  "MongoDBURI": "mongodb://localhost:27017/\
      mongopop?authSource=admin&socketTimeoutMS=\
      30000&maxPoolSize=20",
  "collectionName": "missing"}'
  -i "localhost:3000/pop/countDocs"
  --header "Content-Type:application/json"

HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 129
ETag: W/"81-ct+QBrPOtMmSpo0mjFJ3BA"
Date: Mon, 30 Jan 2017 08:40:49 GMT
Connection: keep-alive

{"success":false,"count":0,"error":"Failed to count\
  the documents: Collection missing does not exist.\
  Currently in strict mode."}
```

The `POST` method for the `pop/sampleDocs` route path works in a very similar way:

```
router.post('/sampleDocs', function(req, res, next)
{
  /* Request from client to read a sample of the
  documents from a collection; the request
  should be of the form:

  {
    MongoDBURI: string; // Connect string for
                        // MongoDB instance
    collectionName: string;
    numberDocs: number; // How many docs should
                        // be in the result set
  }

  The response will contain:

  {
    success: boolean;
    documents: string;  // Sample of documents
    error: string;
  }
  */

  var requestBody = req.body;
  var database = new DB;

  database.connect(requestBody.MongoDBURI)
  .then(
    function() {
        // Returning will pass the promise returned
        // by sampleCollection to the next .then in
        // the chain
        return database.sampleCollection(
            requestBody.collectionName,
            requestBody.numberDocs)
    })  // No function is provided to handle the
        // connection failing and so that
        // error will flow through to the next
        // .then
  .then(
    function(docs) {
        return {
            "success": true,
            "documents": docs,
            "error": ""
            };
    },
    function(error) {
      console.log(
          'Failed to retrieve sample data: ' +
              error);
      return {
              "success": false,
              "documents": null,
              "error":
                "Failed to retrieve sample data: "
                    + error
          };
    })
  .then(
    function(resultObject) {
      database.close();
      res.json(resultObject);
    }
  )
})
```

Testing this new operation:

```
curl -g -X POST --data '{
"MongoDBURI": "mongodb://localhost:27017/mongopop?
    authSource=admin&socketTimeoutMS=30000&
    maxPoolSize=20",
  "collectionName": "simples",
  "numberDocs": "2"
}' "localhost:3000/pop/sampleDocs" --header
    "Content-Type:application/json" |
    python -mjson.tool

  % Total    % Received % Xferd  Average Speed ...
                                 Dload  Upload ...
100   669  100   509  100   160    188     59  ...
{
    "documents": [
        {
            "_id": "5859696971c490c4cacab431",
            "email": "dwilliams8h@arstechnica.com",
            "first_name": "Donald",
            "gender": "Male",
            "id": 306,
            "ip_address": "223.153.70.35",
            "last_name": "Williams",
            "mongopopComment": "MongoPop has been
                here",
            "mongopopCounter": 50
        },
        {
            "_id": "586d36b95c6401b247e4e83d",
            "email": "bgrantat@deliciousdays.com",
            "first_name": "Brian",
            "gender": "Male",
            "id": 390,
            "ip_address": "152.116.123.169",
            "last_name": "Grant",
            "mongopopComment": "MongoPop has been
                here",
            "mongopopCounter": 8
        }
    ],
    "error": "",
    "success": true
}
```

The `POST` method for `pop/updateDocs` is a little more complex as the caller can request multiple update operations be performed. The simplest way to process multiple asynchronous, promise-returning function calls in parallel is to build an array of the tasks and pass it to the `Promise.all` method which returns a promise that either resolves after all of the tasks have succeeded or is rejected if any of the tasks fail:

```
router.post('/updateDocs', function(req, res, next)
  {
    /* Request from client to apply an update to
    all documents in a collection which match a
    given pattern; the request should be of the
    form:

    {
      MongoDBURI: string;     // Connect string
      collectionName: string;
      matchPattern: Object;   // Filter to
                              // determine which
                              // documents should
                              // be updated (e.g.
                              // {"gender": "Male"
                              // })
      dataChange: Object;     // Change to be
                              // applied to each
                              // matching change
                              // (e.g. {"$set":
                              // {"mmyComment":
                              //  "This is a man"},
                              // "$inc": {
                              // "myCounter": 1}}')
      threads: number;        // How many times to
                              // repeat (in
                              // parallel) the
                              // operation
    }

    The response will contain:

    {
      success: boolean;
      count: number;    // The number of docs
                        // updated - should be the
                        // number of docs matching
                        // the pattern multiplied
                        // by the number of threads
      error: string;
    }
    */

    var requestBody = req.body;
    var database = new DB;

    database.connect(requestBody.MongoDBURI)
    .then(
      function() {

        // Build up a list of the operations to
        // be performed

        var taskList = []
        for (var i=0; i < requestBody.threads;
          i++) {
            taskList.push(
              database.updateCollection(
              requestBody.collectionName,
              requestBody.matchPattern,
              requestBody.dataChange));
        }

        // Asynchronously run all of the operations

        var allPromise = Promise.all(taskList);
        allPromise.then(
          function(values) {
              documentsUpdated = values.reduce(add,
                0);
              return {
                "success": true,
                "count": documentsUpdated,
                "error": {}
              };
            },
            function(error) {
              console.log("Error updating documents"
                  + error);
              return {
                "success": false,
                "count": 0,
                "error": "Error updating documents: "
                    + error
              };
            })
            .then(
              function(resultObject) {
                database.close();
                res.json(resultObject);
              }
            )
        },
        function(error) {
          console.log("Failed to connect to the
              database: " + error);
          resultObject = {
            "success": false,
            "count": 0,
            "error": "Failed to connect to the
                database: " + error
          };
          res.json(resultObject);
        }
      );
})
```

Testing with `curl`:

```
curl -g -X POST --data '{
  "MongoDBURI": "mongodb://localhost:27017/\
    mongopop?authSource=admin&socketTimeoutMS=\
    30000&maxPoolSize=20",
  "collectionName": "simples",
  "matchPattern": {
    "gender": "Male"
  },
  "dataChange": {
    "$set": {
      "mongopopComment": "This is a man"
    },
    "$inc": {
      "mongopopCounter": 1
    }
  },
  "threads": "5"
}' "localhost:3000/pop/updateDocs"
    --header "Content-Type:application/json"

{"success":true,"count":6951670,"error":{}}
```

The final method uses example data from a service such as Mockaroo to populate a MongoDB collection. A helper function is created that makes the call to that external service:

```
var request = require("request");

function requestJSON(requestURL) {

  // Retrieve an array of example JSON documents
  // from an external source e.g. mockaroo.com.
  // Returns a promise that either resolves to the
  // results from the JSON service or rejects with
  // the received error.

  return new Promise(function (resolve, reject){

    // Mockaroo can have problems with https
    // - this is random sample data so by
    // definition shouldn't need to be private
    finalDocURL = requestURL.replace('https',
        'http');

    request({url: finalDocURL, json: true},
      function (error, response, body) {
        if (error || response.statusCode != 200) {
          console.log("Failed to fetch documents: "
              + error.message);
          reject(error.message);
        } else {
          resolve(body);
        }
      })
  })
}
```

That function is then used in the `POST` method for `/pop/addDocs`:

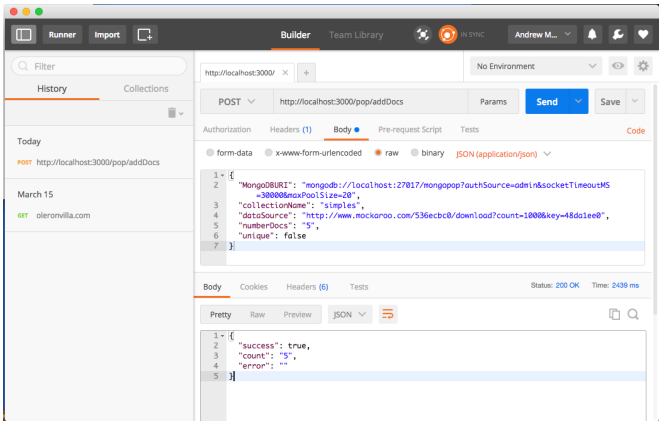
```
router.post('/addDocs', function(req, res, next) {

  /* Request from client to add a number of
  documents to a collection; the request
  should be of the form:

  {
      MongoDBURI: string; // Connect string
      collectionName: string;
      dataSource: string; // e.g. a Mockaroo.com
                          // URL to produce example
                          // docs
      numberDocs: number; // How many (thousands)
                          // documents should be
                          // added
      unique: boolean;    // Whether each batch of
                          // 1,000 documents should
                          // be distinct from the
                          // others (much slower if
                          // set to true)
  }

  The response will contain:

  {
    success: boolean;
    count: number;      // How many documents added
    error: string;
  }
  */

  var requestBody = req.body;
  var uniqueDocs = req.body.unique;
  var batchesCompleted = 0;
  var database = new DB;
  var docURL = requestBody.dataSource;

  database.connect(requestBody.MongoDBURI)
  .then(
    function() {
        if (uniqueDocs) {

        // Need to fetch another batch of unique
        // documents for each batch of 1,000 docs

        for (i = 0; i < requestBody.numberDocs;
            i++) {

          // Fetch the example documents (based on
          // the caller's source URI)

          requestJSON(docURL)
          .then(
            function(docs) {

              // The first function provided as a
              // parameter to "then" is called if
              // the promise is resolved
              // successfully. The "requestJSON"
              // method returns the retrieved
              // documents which the code in this
              // function sees as the "docs"
              // parameter. Write these docs to
              // the database:

              database.popCollection(
                requestBody.collectionName, docs)
              .then(
                function(results) {
                  return batchesCompleted++;
                },
                function(error) {

                  // The second function provided
                  // as a parameter to "then" is
                  // called if the promise is
                  // rejected. "err" is set to
                  // to the error passed by
                  // popCollection

                  database.close();
                  resultObject = {
                    "success": false,
                    "count": batchesCompleted,
                    "error": "Failed to write mock
                     data: " + error
                  };

                  res.json(resultObject);
                  throw(false);
                }
              )
              .then(
                function() {
```

```
            // If all off the batches have
            // been (successfully) added
            // then build and send the
            // response.

            if (batchesCompleted ==
                requestBody.numberDocs) {
              database.close();
              console.log('Wrote all Mock
                data');
              resultObject = {
                "success": true,
                "count": batchesCompleted,
                "error": ""
              };
              res.json(resultObject);
            }
          },
          function(error) {}
          )
          },
          function(error) {
            database.close();
            resultObject = {
              "success": false,
              "count": batchesCompleted,
              "error": "Failed to fetch mock
                  data: " + error
            };
            res.json(resultObject);
          }
        )
      }
  } else {

    // Fetch one set of sample data and then
    // use for repeated batches of writes

    requestJSON(docURL)
    .then(
      function(docs) {

        // Build an array of popCollection
        // calls (not being executed at this
        // point)

        var taskList = [];
        for (i = 0; i < requestBody.numberDocs;
            i++) {
          taskList.push(database.popCollection(
            requestBody.collectionName, docs))
        }

        // Promise.all executes all of the
        // tasks in the provided array
        // asynchronously (i.e. they can run in
        // parallel).

        var allPromise = Promise.all(taskList);
        allPromise.then(
          function(result) {
            database.close();
            resultObject = {
              "success": true,
              "count": requestBody.numberDocs,
              "error": ""
            };
            res.json(resultObject);
          },
          function(error) {
            database.close();
            resultObject = {
              "success": false,
              "count": 0, // If some writes
                          // succeeded then the
                          // real count may be
                          // > 0
              "error": "Failed to write data: "
                  + error
            };
            res.json(resultObject);
          }
        )
      },
      function(error) {
        database.close();
        resultObject = {
         "success": false,
         "count": 0,
         "error": "Failed to fetch mock data: "
         ``+ error
        };
        res.json(resultObject);
      }
    )
  }
},
function(error) {
  resultObject = {
    "success": false,
    "count": 0,
    "error": "Failed to connect to database: "
        + error
  };
  res.json(resultObject);
}
)
})
```

This method is longer than the previous ones – mostly because there are two paths:

- In the first path, the client has requested that a fresh set of 1,000 example documents be used for each pass at adding a batch of documents. This path is much slower and will eat through your Mockaroo quota much faster.
- In the second path, just one batch of 1,000 example documents is fetched from Mockaroo and then those same documents are repeatedly added. This path is faster but it results in duplicate documents (apart from a MongoDB-created `_id` field). This path cannot be used if the `_id` is part of the example documents generated by Mockaroo.

So far, we've used the Chrome browser and the `curl` command-line tool to test the REST API. A third approach is to use the Postman Chrome app.



Debugging tips

One way to debug a Node.js application is to liberally sprinkle `console.log` messages throughout your code but that takes extra effort and bloats your code base. Every time you want to understand something new, you must add extra logging to your code and then restart your application.

Developers working with browser-side JavaScript benefit from the excellent tools built into modern browsers – for example, Google's Chrome Developer Tools which let you:

- Browse code (e.g. HTML and JavaScript)
- Add breakpoints
- View & alter contents of variables
- View and modify CSS styles
- View network messages
- Access the console (view output and issue commands)
- Check security details
- Audit memory use, CPU, etc.

You open the Chrome DevTools within the Chrome browser using "View/Developer/Developer Tools".

Fortunately, you can use the `node-debug` command of `node-inspector` to get a very similar experience for Node.js backend applications. To install `node-inspector`:

```
npm install -g node-inspector
```

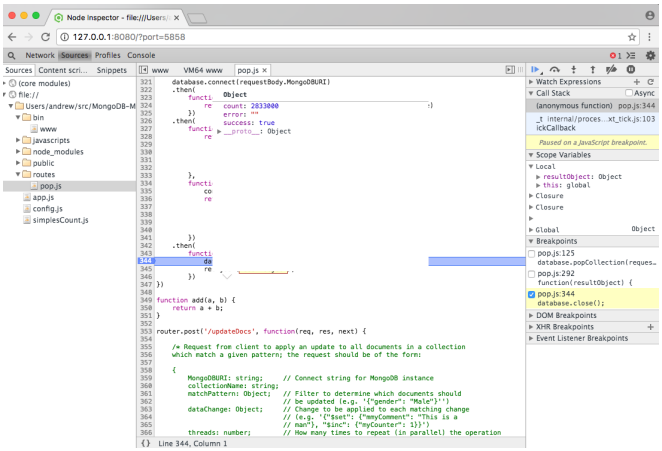
`node-inspector` can be used to debug the Mongopop Express application by starting it with `node-debug` via the `express-debug` script in `package.json`:

```
"scripts": {
    "start": "cd public && npm run tsc && cd .. &&
        node ./bin/www",
    "debug": "cd public && npm run tsc && cd .. &&
        node-debug ./bin/www",
    "tsc": "cd public && npm run tsc",
    "tsc:w": "cd public && npm run tsc:w",
    "express": "node ./bin/www",
    "express-debug": "node-debug ./bin/www",
    "postinstall": "cd public && npm install"
  },
```

To run the Mongopop REST API with `node-debug`, kill the Express app if it's already running and then execute:

```
npm run express-debug
```

Note that this automatically adds a breakpoint at the start of the app and so you will need to skip over that to run the application.



Depending on your version of Node.js, you may see this error:

```
node-debug bin/www
Node Inspector v0.12.8
Visit http://127.0.0.1:8080/?port=5858 to start \
  debugging.
Debugging `bin/www`

Debugger listening on 127.0.0.1:5858
/usr/local/lib/node_modules/node-inspector/lib/\
  InjectorClient.js:111
      cb(error, NM[0].ref);
                     ^

TypeError: Cannot read property 'ref' of undefined
    at InjectorClient.<anonymous> \
      (/usr/local/lib/node_modules/node-inspector/\
        lib/InjectorClient.js:111:22)
    at /usr/local/lib/node_modules/node-inspector/\
      lib/DebuggerClient.js:121:7
    at Object.value (/usr/local/lib/node_modules/\
      node-inspector/lib/callback.js:23:20)
    at Debugger._processResponse (/usr/local/lib/\
      node_modules/node-inspector/lib/debugger.js:95:21)
    at Protocol.execute (_debugger.js:121:14)
    at emitOne (events.js:96:13)
    at Socket.emit (events.js:188:7)
    at readableAddChunk (_stream_readable.js:176:18)
    at Socket.Readable.push (_stream_readable.js:134:10)
    at TCP.onread (net.js:551:20)
```

If you do, apply this patch to `/usr/local/lib/node_modules/node-inspector/lib/InjectorClient.js`.

Chapter summary

This chapter stepped through how to implement a REST API using Express. We also looked at three different ways to test this API and how to debug Node.js applications. This REST API is required by both the Angular and React web app clients, as well as by a number of alternative UIs (including an Amazon Alex Skill).

The next chapter implements the Angular client that makes use of the REST API – by the end of that chapter, you will understand the end-to-end steps required to implement an application using the MEAN stack.

If implementing all of these route sets felt a little repetitive and tedious, then you should look forward to the final chapter, when you'll see how MongoDB Stitch takes care of this for you.

Building a client UI using Angular 2 (formerly AngularJS) & TypeScript

This chapter demonstrates how to use Angular 2 (the evolution of Angular.js) to implement a remote web-app client for the *Mongopop* application.

Downloading, running, and using the Mongopop application

The Angular client code is included as part of the Mongopop package installed earlier.

The backend application should be run in the same way as in previous chapters. The client software needs to be *transpiled* from Typescript to JavaScript – the client software running in a remote browser can then download the JavaScript files and execute them.

The existing `package.json` file includes a script for transpiling the Angular 2 code:

```
  "scripts": {
      ...
    "tsc:w": "cd public && npm run tsc:w",
      ...
},
```

That `tsc:w` delegates the work to a script of the same name defined in `public/package.json`;

```
  "scripts": {
      ...
    "tsc:w": "tsc -w",
      ...
},
```

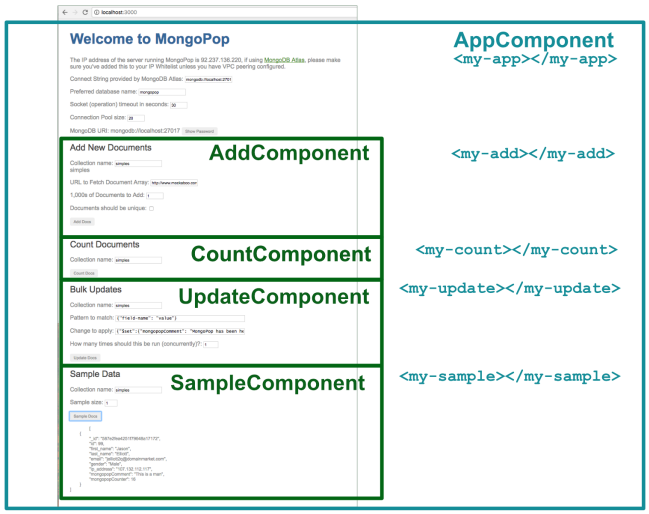
`tsc -w` continually monitors the client app's Typescript files and reruns the transpilation every time they are edited.

To start the continual transpilation of the Angular 2 code:

```
npm run rsc:w
```

Component architecture of the Mongopop Angular UI

Angular applications (both AngularJS and Angular2) are built from one or more, nested components – Mongopop is no exception:



The main component (`AppComponent`) contains the HTML and logic for connecting to the database and orchestrating its sub-components. Part of the definition of `AppComponent` is meta data/decoration to indicate that it should be loaded at the point that a `my-app` element (`<my-app></my-app>`) appears in the `index.html` file (once the component is running, its output replaces whatever holding content sits between `<my-app>` and `</my-app>`). `AppComponent` is implemented by:

- A Typescript file containing the `AppComponent` class (including the data members, initialization code, and member functions
- A HTML file containing
  - HTML layout
  - Rendering of data members
  - Elements to be populated by sub-components
  - Data members to be passed down for use by sub-components
  - Logic (e.g. what to do when the user changes the value in a form)
- (Optionally) a *CSS* file to customize the appearance of the rendered content

Mongopop is a reasonably flat application with only one layer of sub-components below `AppComponent`, but more complex applications may nest deeper.

Changes to a data value by a parent component will automatically be propagated to a child – it's best practice to have data flow in this direction as much as possible. If a data value is changed by a child ***and*** the parent (either directly or as a proxy for one of its other child components) needs to know of the change, then the child triggers an event. That event is processed by a handler registered by the parent – the parent may then explicitly act on the change, but even if it does nothing explicit, the change flows to the other child components.

This table details what data is passed from `AppComponent` down to each of its children and what data change events are sent back up to `AppComponent` (and from there, back down to the other children):

Flow of data between Angular components

Child component	Data passed down	Data changes pased back up
`AddComponent`	Data service	Collection name
	Collection name	
	Mockaroo URL	
`CountComponent`	Data service	Collection name
	Collection name	
`UpdateComponent`	Data service	Collection name
	Collection name	
`SampleComponent`	Data service	Collection name
	Collection name	

What are all of these files?

To recap, the files and folders covered earlier in this book:

- **`package.json`**: Instructs the Node.js package manager (`npm`) what it needs to do; including which dependency packages should be installed
- **`node_modues`**: Directory where `npm` will install packages
- **`node_modues/mongodb`**: The MongoDB driver for Node.js
- **`node_modues/mongodb-core`**: Low-level MongoDB driver library; available for framework developers (application developers should avoid using it directly)

- **`javascripts/db.js`**: A JavaScript module we've created for use by our Node.js apps (in this book, it will be Express) to access MongoDB; this module in turn uses the MongoDB Node.js driver.
- **`config.js`**: Contains the application–specific configuration options
- **`bin/www`**: The script that starts an Express application; this is invoked by the `npm start` script within the `package.json` file. Starts the HTTP server, pointing it to the `app` module in `app.js`
- **`app.js`**: Defines the main backend application module (`app`). Configures:
  - That the application will be run by Express
  - Which routes there will be & where they are located in the file system (`routes` directory)
  - What view engine to use (Jade in this case)
  - Where to find the *views* to be used by the view engine (`views` directory)
  - What middleware to use (e.g. to parse the JSON received in requests)
  - Where the static files (which can be read by the remote client) are located (`public` directory)
  - Error handler for queries sent to an undefined route
- **`views`**: Directory containing the templates that will be used by the Jade view engine to create the HTML for any pages generated by the Express application (for this application, this is just the error page that's used in cases such as mistyped routes ("404 Page not found"))
- **`routes`**: Directory containing one JavaScript file for each Express route
  - **`routes/pop.js`**: Contains the Express application for the `/pop` route; this is the implementation of the Mongopop REST API. This defines methods for all of the supported route paths.
- **`public`**: Contains all of the static files that must be accessible by a remote client (e.g., our Angular to React apps).

Now for the *new files* that implement the Angular client (note that because it must be downloaded by a remote browser, it is stored under the `public` folder):

- **`public/package.json`**: Instructs the Node.js package manager (`npm`) what it needs to do; including which dependency packages should be installed (i.e. the same as `/package.json` but this is for the Angular client app)
- **`public/index.html`**: Entry point for the application; served up when browsing to `http://backend-server/`. Imports `public/system.config.js`
- **`public/system.config.js`**: Configuration information for the Angular client app; in particular defining the remainder of the directories and files:
  - **`public/app`**: Source files for the client application – including the Typescript files (and the transpiled JavaScript files) together the HTML and any custom CSS files. Combined, these define the Angular components.
    - **`public/app/main.ts`**: Entry point for the Angular app. Bootstraps `public/app/app.module.ts`
    - **`public/app/app.module.ts`**: Imports required modules, declares the application components and any services. Declares which component to bootstrap (`AppComponent` which is implemented in `public/app/app.component.*`)
    - **`public/app/app.component.html`**: HTML template for the top-level component. Includes elements that are replaced by sub-components
    - **`public/app/app.component.ts`**: Implements the `AppComponent` class for the top-level component
    - **`public/app/X.component.html`**: HTML template for sub-component *X*
    - **`public/app/X.component.ts`**: Implements the class for sub-component *X*
    - **`AddDocsRequest.ts`**, **`ClientConfig.ts`**, **`CountDocsRequest.ts`**, **`MongoResult.ts`**, **`MongoReadResult.ts`**, **`SampleDocsRequest.ts`**, & **`UpdateDocsRequest.ts`**: Classes that match

the request parameters and response formats of the REST API that's used to access the backend

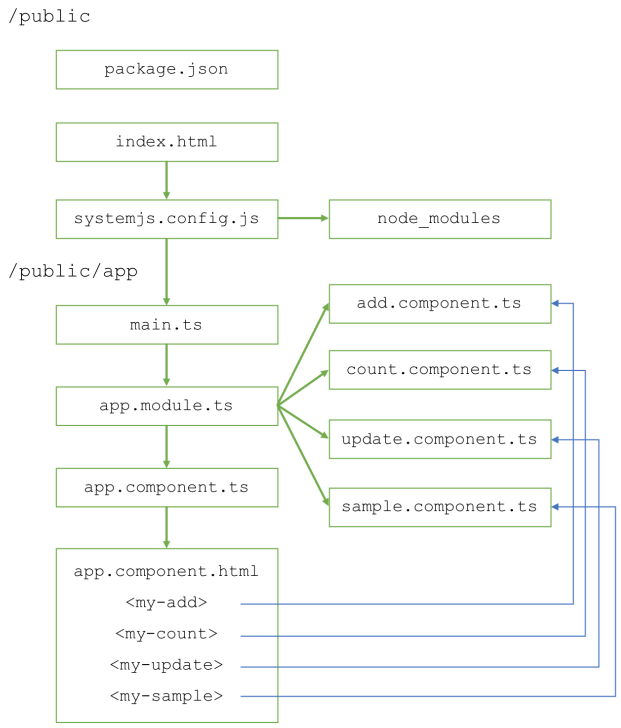
- **`data.service.ts`**: Service used to access the backend REST API (mostly used to access the database)
- **`X.js* & *X.js.map`**: Files which are generated by the transpilation of the Typescript files.

- **`public/node-modules`**: Node.js modules used by the Angular app (as opposed to the Express, server-side Node.js modules)
- **`public/styles.css`**: CSS style sheet (imported by `public/index.html)` – applies to all content in the home page, not just content added by the components
- **`public/stylesheets/styles.css`**: CSS style sheet (imported by `public/app/app.component.ts` and the other components) – note that each component could have their own, specialized style sheet instead

"Boilerplate" files and how they get invoked

This is an imposing number of new files and this is one of the reasons that Angular is often viewed as the more complex layer in the application stack. One of the frustrations for many developers, is the number of files that need to be created and edited on the client side before your first line of component/application code is executed. The good news is that there is a consistent pattern and so it's reasonable to fork you app from an existing project – the Mongopop app can be cloned from GitHub or, the Angular QuickStart can be used as your starting point.

As a reminder, here is the relationship between these common files (and our application-specific components):



Contents of the "boilerplate" files

This section includes the contents for each of the non-component files and then remarks on some of the key points.

public/package.json

```
{
  "name": "MongoPop",
  "version": "0.1.1",
  "description": "Mongopop client - add data sets and \
    run traffic on MongoDB",
  "scripts": {
    "start": "tsc && concurrently \"tsc -w\" ",
    "postinstall": "typings install",
    "tsc": "tsc",
    "tsc:w": "tsc -w",
    "typings": "typings"
  },
  "keywords": [],
  "author": "Andrew Morgan",
  "license": "ISC",
  "dependencies": {
    "@angular/common": "2.0.0-rc.6",
    "@angular/compiler": "2.0.0-rc.6",
    "@angular/core": "2.0.0-rc.6",
    "@angular/forms": "2.0.0-rc.6",
    "@angular/http": "2.0.0-rc.6",
    "@angular/platform-browser": "2.0.0-rc.6",
    "@angular/platform-browser-dynamic": "2.0.0-rc.6",
    "@angular/router": "3.0.0-rc.2",
    "@angular/upgrade": "2.0.0-rc.6",
    "bootstrap": "^3.3.6",
    "core-js": "^2.4.1",
    "gulp": "^3.9.1",
    "reflect-metadata": "^0.1.3",
    "rx": "^4.1.0",
    "rxjs": "5.0.0-beta.11",
    "systemjs": "0.19.27",
    "zone.js": "^0.6.17"
  },
  "devDependencies": {
    "concurrently": "^2.2.0",
    "typescript": "^1.8.10",
    "typings": "^1.0.4",
    "canonical-path": "0.0.2",
    "http-server": "^0.9.0",
    "lodash": "^4.11.1",
    "rimraf": "^2.5.2"
  },
  "repository": {}
}
```

The `scripts` section defines what `npm` should do when you type `npm run command-name` from the command line. Of most interest is the `tsc:w` script – this is how the transpiler is launched. After transpiling all of the `.ts` Typescript files, it watches them for changes – retranspiling as needed.

Note that the `dependencies` are for this Angular client. They will be installed in `public/node_modules` when `npm install` is run (for Mongopop, this is done automatically when building the full project).

public/index.html

```
<!DOCTYPE html>
<html>
  <!--
  This is the file that will be served to anyone
  browsing to
  http://server-running-mongopop-backend/
  Then, by loading 'systemjs.config.js', Angular2
  will replace the "my-app" element with the
  Mongopopclient application.
  -->

  <head>
    <!-- <title>MongoPop</title> -->
    <meta charset="UTF-8">
    <meta name="viewport"
      content="width=device-width,
      initial-scale=1">
    <link rel="stylesheet" href="styles.css">

    <!-- Polyfill(s) for older browsers -->
    <script src=
      "node_modules/core-js/client/shim.min.js">
    </script>

    <script
      src="node_modules/zone.js/dist/zone.min.js">
    </script>
    <script
       src=
        "node_modules/reflect-metadata/Reflect.js">
    </script>
    <script
    src="node_modules/systemjs/dist/system.src.js">
    </script>
    <script src="systemjs.config.js"></script>
    <script>
      System.import('app').catch(function(err){
          console.error(err); });
    </script>
  </head>

  <body>
    <my-app>Loading MongoPop client app...</my-app>
  </body>
</html>
```

Focusing on the key lines, the application is started using the `app` defined in `systemjs.config.js`:

```
<script src="systemjs.config.js"></script>
<script>
  System.import('app').catch(function(err){
        console.error(err); });
</script>
```

And the output from the application replaces the placeholder text in the `my-app` element:

```
<my-app>Loading MongoPop client app...</my-app>
```

public/systemjs.config.js

```
/*
 System configuration for Mongopop Angular 2 client
 app
*/
(function (global) {
  System.config({
    paths: {
      // paths serve as alias
      'npm:': 'node_modules/'
    },
    // map tells the System loader where to look
    // for things
    map: {
      // The Mongopop app is within the app folder
      app: 'app',

      // angular bundles
      '@angular/core':
        'npm:@angular/core/bundles/core.umd.js',
      '@angular/common':
        'npm:@angular/common/bundles/common.umd.js',
      '@angular/compiler':
        'npm:@angular/compiler/bundles/\
        compiler.umd.js',
      '@angular/platform-browser':
        'npm:@angular/platform-browser/bundles/\
          platform-browser.umd.js',
      '@angular/platform-browser-dynamic':
        'npm:@angular/platform-browser-dynamic/\
        bundles/platform-browser-dynamic.umd.js',
      '@angular/http':
        'npm:@angular/http/bundles/http.umd.js',
      '@angular/router':
        'npm:@angular/router/bundles/router.umd.js',
      '@angular/forms':
        'npm:@angular/forms/bundles/forms.umd.js',

      // other libraries
      'rxjs':                      'npm:rxjs',
    },
    // packages tells the System loader how to load
    // when no filename and/or no extension
    packages: {
      app: {
        // app/main.js (built from app/main.ts) is
        // the entry point for the Mongopop client
        // app
        main: './main.js',
        defaultExtension: 'js'
      },
      rxjs: {
        defaultExtension: 'js'
      }
    }
  });
})(this);
```

`packages.app.main` is mapped to `public/app/main.js` – note that `main.js` is referenced rather than `main.ts` as it is always the transpiled code that is executed. This is what causes `main.ts` to be run.

public/app/main.ts

```
// Invoked from `system.config.js`; loads
// `app.module.js` (built from `app.module.ts`) and
// then bootstraps the module contained in that
// file (`AppModule`)

import { AppModule } from './app.module';

platformBrowserDynamic().bootstrapModule(
    AppModule);
```

This simply imports and bootstraps the `AppModule` class from `public/app/app.module.ts` (actually `app.module.js`)

public/app/app.module.ts

```
// This is the main Angular2 module for the
// Mongopop client app. It is bootstrapped
// from `main.js` (built from `main.ts`)

import {NgModule} from '@angular/core';
import {BrowserModule} from
    '@angular/platform-browser';
import {HttpModule} from '@angular/http';
import {FormsModule} from '@angular/forms';


// Components making up the Mongopop Angular2
// client app
import {AppComponent} from './app.component';
import {SampleComponent} from './sample.component';
import {AddComponent} from './add.component';
import {CountComponent} from './count.component';
import {UpdateComponent} from './update.component';

// Service for accessing the Mongopop (Express)
// server API
import { DataService } from './data.service';

@NgModule({

  imports: [
  BrowserModule,
  FormsModule,
  HttpModule
  ],

  declarations: [
  AppComponent,
  SampleComponent,
  AddComponent,
  CountComponent,
  UpdateComponent
  ],

  providers: [
  DataService
  ],

    bootstrap: [AppComponent]
  })

export class AppModule { }
```

This is the first file to actually reference the components which make up the Mongopop application!

Note that `NgModule` is the core module for Angular and must always be imported; for this application `BrowserModule`, `HttpModule`, and `FormsModule` are also needed.

The `import` commands also bring in the (`.js`) files for each of the components as well as the data service.

Following the imports, the `@NgModule` *decorator function* takes a JSON object that tells Angular how to run the code for this module (`AppModule`) – including the list of imported modules, components, and services as well as the module/component needed to bootstrap the actual application (`AppComponent`).

Typescript & Observables (before getting into component code)

The most recent, widely supported version is ECMAScript 6 – normally referred to as ES6. ES6 is supported by recent versions of Chrome, Opera, Safari, and Node.js). Some platforms (e.g. Firefox and Microsoft Edge) do not yet support all features of ES6. These are some of the key features added in ES6:

- Classes & modules
- Promises – a more convenient way to handle completion or failure of synchronous function calls (compared to callbacks)
- Arrow functions – a concise syntax for writing function expressions
- Generators – functions that can yield to allow others to execute
- Iterators
- Typed arrays

Typescript is a superset of ES6 (JavaScript); adding static type checking. Angular 2 is written in Typescript and Typescript is the primary language to be used when writing code to run in Angular 2.

Because ES6 and Typescript are not supported in all environments, it is common to *transpile* the code into an earlier version of JavaScript to make it more portable. `tsc` is used to transpile Typescript into JavaScript.

And of course, JavaScript is augmented by numerous libraries. The Mongopop Angular 2 client uses **Observables** from the RxJS reactive libraries which greatly simplify making asynchronous calls to the backend (a pattern historically referred to as AJAX).

RxJS Observables fulfill a similar role to ES6 promises in that they simplify the code involved with asynchronous function calls (removing the need to explicitly pass callback functions). Promises are more contained than Observables, they make a call and later receive a single signal that the asynchronous activity triggered by the call succeeded or failed. Observables can have a more complex lifecycle, including the caller receiving multiple sets of results and the caller being able to cancel the Observable.

The Mongopop application uses two simple patterns when calling functions that return an Observable; the first is used within the components to digest the results from our own data service:

```
var _this = this;

  myLibrary.myAsyncFunction(myParameters)
  .subscribe(
    myResults => {
      // If the observable emits successful results
      // then the first of the arrow functions
      // passed to the `subscribe` method is
      // invoked. `myResults` is an arbitrary name
      // and it is set to the result data sent back
      // by the observable returned by
      // `myAsyncFunction`. If the observable emits
      // multiple result sets then this function is
      // invoked multiple times.

      doSomething(myResults);
      _this.resultsReceived++;
    },
    myError => {
      // If the observable finds a problem then the
      // second of the arrow functions passed to
      // the `subscribe` method is invoked.
      // `myError` is an arbitrary name and it is
      // set to the error data sent back by the
       //observable returned by `myAsyncFunction`.
       // There will be no further results from the
       // observable after this error.

      console.log("Hit a problem: " +
            myError.message);
    },
    () => {
      // Invoked when the observable has emitted
      // everything that it plans to (and
      // no error were found)

      console.log("Finished; received " +
            _this.resultsReceived +
            " sets of results");
    }
  )
```

In Mongopop's use of Observables, we don't have anything to do in the final arrow function and so don't use it (and so it could have used the second pattern instead – but it's interesting to see both).

The second pattern is used within the data service when making calls to the Angular 2 `http` module (this example also shows how we return an Observable back to the components):

```
fetchServerIP() : Observable<string> {

  // This method returns an Observable which
  // resolves to a string

  // Make a http call which returns an Observable
  return this.http.get(this.baseURL + "ip")
  .map(
    // This is called if/when the Observable
    // returned by `http.get` has results for
    // us. Map the response so that this
    // method's returned Observable only
    // supplies the part of the result that the
    // caller cares about.
    response => response.json().ip
  )
  .catch(
    // This is invoked if/when the Observable
    // returned by `http.get` flags an error.
    // Throw an error to the subscriber of the
    // Observable returned by this method.
    (error:any) => Observable.throw(
        error.json().error || 'Server error')
  )
}
```

Calling the REST API

The `DataService` class hides the communication with the backend REST API; serving two purposes:

- Simplifying all of the components' code
- Shielding the components' code from any changes in the REST API signature or behavior – that can all be handled within the DataService

By adding the `@Injectable` decorator to the class definition, any member variables defined in the arguments to the class constructor function will be automatically instantiated (i.e. there is no need to explicitly request a `new Http` object):

```
import {Injectable, OnInit} from '@angular/core';
import {Http, Response, Headers, RequestOptions}
    from '@angular/http';
import {Observable, Subscription} from 'rxjs/Rx';

import 'rxjs/add/operator/map';
import 'rxjs/add/operator/catch';

import {MongoResult} from './MongoResult';
import {ClientConfig} from './ClientConfig';
import {AddDocsRequest} from './AddDocsRequest';
import {SampleDocsRequest} from
    './SampleDocsRequest';
import {MongoReadResult} from './MongoReadResult';
import {UpdateDocsRequest} from
    './UpdateDocsRequest';
import {CountDocsRequest} from
    './CountDocsRequest';

@Injectable()
export class DataService {

  private MongoDBURI: string;
  private baseURL: string =
      "http://localhost:3000/pop/";
          // The URL for the Mongopop service

  constructor (private http: Http) {
  }
```

After the constructor has been called, methods within the class can safely make use of the `http` data member.

Most of the methods follow a very similar pattern and so only a few are explained here; refer to the `DataService` class to review the remainder.

The simplest method retrieves a count of the documents for a given collection:

```
sendCountDocs(CollName: string) :
    Observable<MongoResult> {

  /*
  Use the Mongopop API to count the number of
  documents in the specified collection. It returns
  an Observable that delivers objects of type
  MongoResult.
  */

  // Need to indicate that the request parameters
  // will be in the form of a JSON document
  let headers = new Headers({
    'Content-Type': 'application/json' });
  let options = new RequestOptions({
    headers: headers});

  // The CountDocsRequest class contains the same
  // elements as the `pop/count` REST API POST
  // method expects to receive
  let countDocsRequest = new CountDocsRequest (
    this.MongoDBURI, CollName);
  let url: string = this.baseURL + "countDocs";

  return this.http.post(url, countDocsRequest,
    options)
  .timeout(360000, new Error('Timeout exceeded'))
  .map(response => response.json())
  .catch((error:any) => {
      return Observable.throw(error.toString() ||
        ' Server error')
  });
};
```

This method returns an Observable, which in turn delivers an object of type `MongoResult`. `MongoResult` is defined in `MongoResult.ts`:

```
export class MongoResult {
  success: boolean;
  count: number;
  error: string;

  constructor(success: boolean, count?: number,
      error?: string) {
    this.success = success;
    this.count = count;
    this.error = error;
  }
}
```

The `pop/count` PUT method expects the request parameters to be in a specific format (see earlier table); to avoid coding errors, another Typescript class is used to ensure that the correct parameters are always included – `CountDocsRequest`:

```
export class CountDocsRequest {
  MongoDBURI: string;
  collectionName: string;

  constructor(MongoDBURI?: string,
      collectionName?: string) {
    this.MongoDBURI = MongoDBURI;
    this.collectionName = collectionName;
  }
}
```

`http.post` returns an Observable. If the Observable achieves a positive outcome then the `map` method is invoked to convert the resulting data (in this case, simply parsing the result from a JSON string into a Typescript/ JavaScript object) before automatically passing that updated result through this method's own returned Observable.

The `timeout` method causes an error if the HTTP request doesn't succeed or fail within 6 minutes.

The `catch` method passes on any error from the HTTP request (or a generic error if `error.toString()` is `null`) if none exists.

The `updateDBDocs` method is a little more complex – before sending the request, it must first parse the user-provided strings representing:

- The pattern identifying which documents should be updated
- The change that should be applied to each of the matching documents

This helper function is used to parse the (hopefully) JSON string:

```
tryParseJSON (jsonString: string): Object{

  /*
  Attempts to build an object from the supplied
  string. Raises an error if the conversion fails
  (e.g. if it isn't valid JSON format).
  */

  try {
    let myObject = JSON.parse(jsonString);

    if (myObject && typeof myObject ===
        "object") {
      return myObject;
    }
  }
  catch (error) {
    let errorString = "Not valid JSON: " +
      error.message;
    console.log(errorString);
    new Error(errorString);
  }
  return {};
};
```

If the string is a valid JSON document then `tryParseJSON` returns an object representation of it; if not then it returns an error.

A new class (`UpdateDocsRequest`) is used for the update request:

```
export class UpdateDocsRequest {
  MongoDBURI: string;
  collectionName: string;
  matchPattern: Object;
  dataChange: Object;
  threads: number;

  constructor(MongoDBURI?: string,
      collectionName?: string,
      matchPattern?: Object,
      dataChange?: Object, threads?: number) {
    this.MongoDBURI = MongoDBURI;
    this.collectionName = collectionName;
    this.matchPattern = matchPattern;
    this.dataChange = dataChange;
    this.threads = threads;
  }
}
```

`updateDBDocs` is the method that is invoked from the component code:

```
updateDBDocs (collName: string,
    matchPattern: string, dataChange: string,
    threads: number): Observable<MongoResult> {

  /*
  Apply an update to all documents in a collection
  which match a given pattern. Uses the MongoPop
  API. Returns an Observable which either resolves
  to the results of the operation or throws an
  error.
  */
  let matchObject: Object;
  let changeObject: Object;

  try {
    matchObject = this.tryParseJSON(matchPattern);
  }
  catch (error) {
    let errorString = "Match pattern: " +
      error.message;
    console.log(errorString);
    return Observable.throw(errorString);
  }

  try {
    changeObject = this.tryParseJSON(dataChange);
  }
  catch (error) {
    let errorString = "Data change: " +
      error.message;
    console.log(errorString);
    return Observable.throw(errorString);
  }

  let updateDocsRequest = new UpdateDocsRequest (
    this.MongoDBURI, collName, matchObject,
    changeObject, threads);

  return this.sendUpdateDocs(updateDocsRequest)
  .map(results => {return results})
  .catch((error:any) => {
    return Observable.throw(error.toString() ||
      ' Server error')
  })
}
```

After converting the received string into objects, it delegates the actual sending of the HTTP request to `sendUpdateDocs`:

```
sendUpdateDocs(doc: UpdateDocsRequest) :
    Observable<MongoResult> {
  let headers = new Headers({
    'Content-Type': 'application/json' });
  let options = new RequestOptions({
      headers: headers});
  let url: string = this.baseURL + "updateDocs";

  return this.http.post(url, doc, options)
  .timeout(360000000,
    new Error('Timeout exceeded'))
  .map(response => response.json())
  .catch((error:any) => {
    return Observable.throw(error.toString() ||
      " Server error")
  });
};
```

A simple component that accepts data from its parent

Recall that the application consists of five components: the top-level application which contains each of the add, count, update, and sample components.

When building a new application, you would typically start by designing the the top-level container and then work downwards. As the top-level container is the most complex one to understand, we'll start at the bottom and then work up.

A simple sub-component to start with is the count component:

`public/app/count.component.html` defines the elements that define what's rendered for this component:

```
<h2>Count Documents</h2>
<p>
  Collection name:
  <input #CountCollName
    id="count-collection-name" type="text"
    value="{{MongoDBCollectionName}}">
</p>
<p>
  <button (click)="countDocs(CountCollName.value)">
    Count Docs</button>
</p>
<p>
  <span class="successMessage">{{DocumentCount}}
    </span>
  <span class="errorMessage">{{CountDocError}}
    </span>
</p>
```

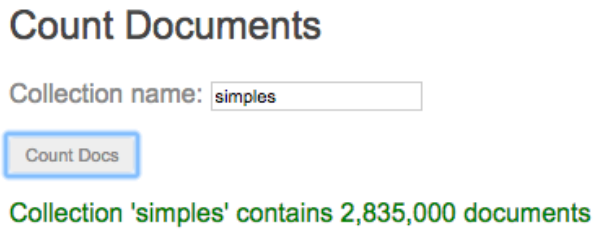
You'll recognize most of this as standard HTML code.

The first Angular extension is for the single `input` element, where the initial `value` (what's displayed in the input box) is set to `{{MongoDBCollectionName}}`. Any name contained within a double pair of braces refers to a data member of the component's class (`public/app/count.component.ts`).

When the button is clicked, `countDocs` (a method of the component's class) is invoked with `CountCollName.value` (the current contents of the input field) passed as a parameter.

Below the button, the class data members of `DocumentCount` and `CountDocError` are displayed – nothing is actually rendered unless one of these has been given a non-empty value. Note that these are placed below the button in the code, but they would still display the resulting values if they were moved higher up – position within the HTML file doesn't impact logic flow. Each of those messages is given a class so that they can be styled differently within the component's CSS file:

```
.errorMessage {
  font-weight: bold;
  color: red;
}

.warningMessage {
  color: brown;
}
.successMessage {
  color: green;
}
```



The data and processing behind the component is defined in `public/app/count.component.ts`:

```
import { Component, OnInit, Injectable,
    EventEmitter, Input, Output}
    from '@angular/core';
import { Response } from '@angular/http';
import { Observable, Subscription } from 'rxjs/Rx';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/catch';

import {DataService} from './data.service';

// This component will be loaded into the
// <my-count> element of `app/app.component.html`

@Component({
  selector: 'my-count',
  templateUrl: 'app/count.component.html',
  styleUrls:  ['stylesheets/style.css']
})

@Injectable()
export class CountComponent implements OnInit {

  CountDocError: string = "";
  DocumentCount: string = "";

  // Parameters sent down from the parent component
  // (AppComponent)
  @Input() dataService: DataService;
  @Input() MongoDBCollectionName: string;

  // Event emitters to pass changes back up to the
  // parent component
  @Output() onCollection =
      new EventEmitter<string>();

  ngOnInit() {}

  // Invoked from the component's html code
  countDocs(CollName: string) {
    this.DocumentCount = "";
    this.CountDocError = "";

    this.dataService.sendCountDocs(CollName)
    .subscribe(
      results => {
      // Invoked if/when the observable is
      // successfully resolved
      if (results.success) {
        this.DocumentCount = "Collection '" +
          CollName + "' contains " +
          results.count.toLocaleString() +
          " documents";
        this.MongoDBCollectionName = CollName;
        this.onCollection.emit(
          this.MongoDBCollectionName);
      }
      else {
        // Invoked if/when the backend successfully
        // sends a response but that response
        // indicates an application-level error
        this.CountDocError = "Application Error: "
          + results.error;
      }
    },
    error => {
      // Invoked if/when the observable throws an
      // error
      this.CountDocError = "Network Error: " +
        error;
    })
  }
}
```

Starting with the `@component` decoration for the class:

```
@Component({
  selector: 'my-count',
  templateUrl: 'app/count.component.html',
  styleUrls:  ['stylesheets/style.css']
})
```

This provides meta data for the component:

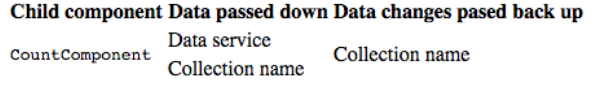
- `selector`: The position of the component within the parent's HTML should be defined by a `<my-count></my-count>` element.
- `templateUrl`: The HMTL source file for the template (`public/app/count.component.ts` in this case – `public` is dropped as the path is relative)
- `styleUrls`: The CSS file for this component – all components in this application reference the same file: `public/stylesheets/style.css`

The class definition declares that it implements the `OnInit` interface; this means that its `ngOnInit()` method will be called after the browser has loaded the component; it's a good place to perform any initialization steps. In this component, it's empty and could be removed.

The two data members used for displaying success/failure messages are initialized to empty strings:

```
this.DocumentCount = "";
this.CountDocError = "";
```

Recall that data is passed back and forth between the count component and its parent:



To that end, two class members are inherited from the parent component – indicated by the `@Input()` decoration:

```
// Parameters sent down from the parent component
// (AppComponent)
@Input() dataService: DataService;
@Input() MongoDBCollectionName: string;
```

The first is an instance of the data service (which will be used to request the document count); the second is the collection name that we used in the component's HTML code. Note that if either of these are changed in the parent component then the instance within this component will automatically be updated.

When the name of the collection is changed within this component, the change needs to be pushed back up to the parent component. This is achieved by declaring an event emitter (`onCollection`):

```
@Output() onCollection =
  new EventEmitter<string>();
...
this.onCollection.emit(this.MongoDBCollectionName);
```

Recall that the HTML for this component invokes a member function: `countDocs(CountCollName.value)` when the button is clicked; that function is implemented in the component class:

```
countDocs(CollName: string) {
  this.DocumentCount = "";
  this.CountDocError = "";

  this.dataService.sendCountDocs(CollName)
  .subscribe(
    results => {
    // Invoked if/when the observable is
    // successfully resolved
    if (results.success) {
      this.DocumentCount = "Collection '" +
        CollName + "' contains " +
        results.count.toLocaleString() +
        " documents";
      this.MongoDBCollectionName = CollName;
      this.onCollection.emit(
        this.MongoDBCollectionName);
    }
    else {
      // Invoked if/when the backend sucessfull
      //y sends a response but that response
      // indicates an application-level error
      this.CountDocError = "Application Error: " +
        results.error;
    }
},
error => {
  // Invoked if/when the observable throws an error
  this.CountDocError = "Network Error: " + error;
  })
}
```

After using the data service to request the document count, either the success or error messages are sent – depending on the success/failure of the requested operation. Note that there are two layers to the error checking:

1. Was the network request successful? Errors such as a bad URL, out of service backend, or loss of a network connection would cause this check to fail.
2. Was the backend application able to execute the request successfully? Errors such as a non-existent collection would cause this check to fail.

Note that when `this.CountDocError` or `this.DocumentCount` are written to, Angular will automatically render the new values in the browser.

Passing data down to a sub-component (and receiving changes back)

We've seen how `CountComponent` can accept data from its parent and so the next step is to look at that parent – `AppComponent`.

The HTML template `app.component.html` includes some of its own content, such as collecting database connection information, but most of it is delegation to other components. For example, this is the section that adds in `CountComponent`:

```
<div>
  <my-count
    [dataService]="dataService"
    [MongoDBCollectionName]="MongoDBCollectionName"
    (onCollection)="onCollection($event)">
  </my-count>
</div>
```

Angular will replace the `<my-count></my-count>` element with `CountComponent`; the extra code within that element passes data down to that sub-component. For passing data members down, the syntax is:

```
[name-of-data-member-in-child-component]=
  "name-of-data-member-in-this-component"
```

As well as the two data members, a reference to the `onCollection` event handler is passed down (to allow `CountComponent` to propagate changes to the collection name back up to this component). The syntax for this is:

```
(name-of-event-emitter-in-child-component)=
  "name-of-event-handler-in-this-component($event)"
```

As with the count component, the main app component has a Typescript class – defined in `app.component.ts` – in addition to the HTML file. The two items that must be passed down are the data service (so that the count component can make requests of the backend) and the collection name – these are both members of the `AppComponent` class.

The `dataService` object is implicitly created and initialized because it is a parameter of the class's constructor, and because the class is decorated with `@Injectable`:

```
import { DataService } from './data.service';
...
@Injectable()
export class AppComponent implements OnInit {
  ...
  constructor (private dataService: DataService) {}
  ...
}
```

`MongoDBCollectionName` is set during component initialization within the `ngOnInit()` method by using the data service to fetch the default client configuration information from the backend:

```
export class AppComponent implements OnInit {
  ...
  MongoDBCollectionName: string;
  ...
  ngOnInit() {

    // Fetch the default client config from the
    // backend

    this.dataService.fetchClientConfig().subscribe(
    results => {
      // This code is invoked if/when the
      // observable is resolved successfully
      ...
      this.MongoDBCollectionName =
        results.mongodb.defaultCollection;
      ...
    },
    error => {
      // This code is executed if/when the
      // observable throws an error.
      console.log("Failed to fetch client content"+
        data. Reason: " + error.toString);
    });
  }
  ...
}
```

Finally, when the collection name is changed in the count component, the event that it emits gets handled by the event handler called, `onCollection`, which uses the new value to update its own data member:

```
// This is invoked when a sub-component emits an
// onCollection event to indicate that the user has
// changes the collection within its form.
// The binding is created in app.component.html
onCollection(CollName: string) {
  this.MongoDBCollectionName = CollName;
}
```

Conditionally including a component

It's common that a certain component should only be included if a particular condition is met. Mongopop includes a feature to allow the user to apply a bulk change to a set of documents - selected using a pattern specified by the user. If they don't know the typical document structure for the collection then it's unlikely that they'll make a sensible change. Mongopop forces them to first retrieve a sample of the documents before they're given the option to make any changes.

The `ngIf` directive can be placed within the opening part of an element (in this case a `<div>`) to make that element conditional. This approach is used within `app.component.html` to only include the update component if the `DataToPlayWith` data member is TRUE:

```
<div *ngIf="DataToPlayWith">
  <my-update
    [dataService]="dataService"
    [MongoDBCollectionName]="MongoDBCollectionName"
    (onCollection)="onCollection($event)">
  </my-update>
</div>
```

Note that, as with the count component, if the update component is included then it's passed the data service and collection name and that it also passes back changes to the collection name.

Angular includes other directives that can be used to control content; `ngFor` being a common one as it allows you to iterate through items such as arrays:

```
<ul>
  <li *ngFor="let item of items; let i = index">
    {{i}} {{item}}
  </li>
</ul>
```

`DataToPlayWith` is initialized to FALSE in the `AppComponent` class and then set to TRUE when an event is received by the `onSample` event handler:

```
export class AppComponent implements OnInit {
  ...
  DataToPlayWith: boolean = false;
  ...
  // This is invoked when the sub-component
  // (SampleComponent from sample.component.ts)
  // emits an onSample event. The binding is
  // created in app.component.html
  onSample(haveSampleData: boolean) {
    // Expose the UpdateComponent
    this.DataToPlayWith = haveSampleData;
  }
  ...
}
```

Returning to `app.component.html`, an extra handler (`onSample`) is passed down to the sample component:

```
<div>
  <my-sample
    [dataService]="dataService"
    [MongoDBCollectionName]="MongoDBCollectionName"
    (onSample)="onSample($event)"
    (onCollection)="onCollection($event)">
  </my-sample>
</div>
```

`sample.component.html` is similar to the HTML code for the count component but there is an extra input for how many documents should be sampled from the collection:

```
<h2>Sample Data</h2>
<p>
  Collection name:
  <input #SampleCollName
    id="sample-collection-name" type="text"
    value="{{MongoDBCollectionName}}">
</p>
<p>
  Sample size:
    <input #SampleSize id="sample-size"
      type="number" min="1" max="10" value="1"/>
</p>
<p>
  <button (click)="sampleDocs(SampleCollName.value,
    SampleSize.value)">Sample Docs</button>
</p>
<p>
  <span class="errorMessage"> {{SampleDocError}}
  </span>
</p>
<div class="json">
  <pre>
    {{SampleDocResult}}
  </pre>
</div>
```

On clicking the button, the collection name and sample size are passed to the `sampleDocs` method in `sample.component.ts` which (among other things) emits an event back to the `AppComponent`'s event handler using the `onSample` event emitter:

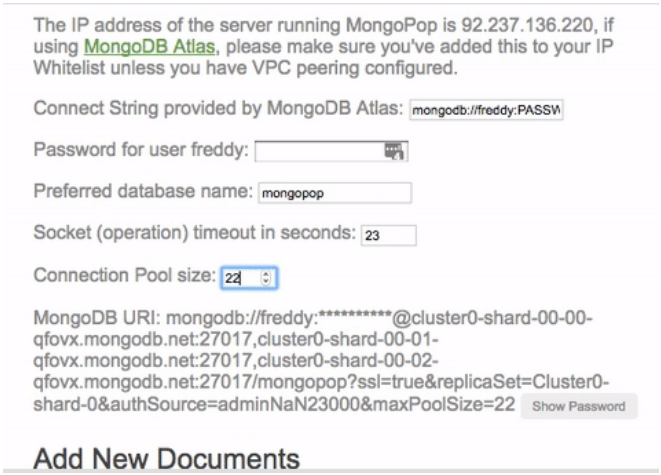
```
@Injectable()
export class SampleComponent implements OnInit {
  ...
  // Parameters sent down from the parent component
  // (AppComponent)
  @Input() dataService: DataService;
  @Input() MongoDBCollectionName: string;

  // Event emitters to pass changes back up to the
  // parent component
  @Output() onSample=new EventEmitter<boolean>();
  @Output() onCollection=new
    EventEmitter<string>();
  ...
  sampleDocs(CollName: string, NumberDocs: number)
  {
    this.SampleDocResult = "";
    this.SampleDocError = "";
    this.onSample.emit(false);

    this.dataService.sendSampleDoc(CollName,
      NumberDocs)
    .subscribe(
      results => {
        // Invoked if/when the observable is
        // successfully resolved
        if (results.success) {
          this.SampleDocResult =
           this.syntaxHighlight(results.documents);
          this.MongoDBCollectionName = CollName;
          this.onSample.emit(true);
          this.onCollection.emit(
            this.MongoDBCollectionName);
        } else {
          this.SampleDocError =
            "Application Error: " + results.error;
        }
      },
      error => {
        // Invoked if/when the observable throws an
        // error
        this.SampleDocError = "Network Error: " +
          error.toString;
      }
    );
  }
}
```

Other code highlights

Returning to `app.component.html`; there is some content there in addition to the sub-components:

```
<h1>Welcome to MongoPop</h1>
<div>
  <p>
      The IP address of the server running MongoPop
      is {{serverIP}}, if using
      <a href="https://cloud.mongodb.com"
        name="MongoDB Atlas" target="_blank">
        MongoDB Atlas</a>,
      please make sure you've added this to your IP
      Whitelist unless you have VPC peering
      configured.
  </p>
</div>

<div>
  <p>
    Connect String provided by MongoDB Atlas:
    <input #MongoDBBaseString
      id="MongoD-base-string"
     value="{{dBInputs.MongoDBBaseURI}}"
     (keyup)="setBaseURI(MongoDBBaseString.value)"
     (change)="setBaseURI(MongoDBBaseString.value)"
     />
  </p>

  <!-- Only ask for the password if the MongoDB URI
  has been changed from localhost -->
  <div *ngIf="dBInputs.MongoDBUser">
    <p>
        Password for user {{dBInputs.MongoDBUser}}:
        <input #MongoDBPassword
          id="MongoDB-password"
          value="{{dBInputs.MongoDBUserPassword}}"
          type="password"
          (keyup)="setPassword(
            MongoDBPassword.value)"
          (change)="setPassword(
            MongoDBPassword.value)"/>
    </p>
  </div>
  <p>
    Preferred database name:
    <input #MongoDBDBName id="MongoDB-db-name"
      value="{{dBInputs.MongoDBDatabaseName}}"
      (keyup)="setDBName(MongoDBDBName.value)"
      (change)="setDBName(MongoDBDBName.value)"/>
  </p>
  <p>
    Socket (operation) timeout in seconds:
    <input #SocketTimeout id="socket-timeout"
      value="{{dBInputs.MongoDBSocketTimeout}}"
      type="number" min="1" max="1000"
      (change)="setMongoDBSocketTimeout(
        SocketTimeout.value)"/>
  </p>
  <p>
    Connection Pool size:
    <input #ConnectionPoolSize
    id="connection-pool-size"
    value="{{dBInputs.MongoDBConnectionPoolSize}}"
    type="number" min="1" max="1000"
    (change)="setMongoDBConnectionPoolSize(
      ConnectionPoolSize.value)"/>
  </p>
  <p>
    MongoDB URI: {{dBURI.MongoDBURIRedacted}}
    <button (click)="showPassword(true)">
      Show Password</button>
  </p>
  ...
</div>
```

Most of this code is there to allow a full MongoDB URI/connection string to be built based on some user-provided attributes. Within the input elements, two event types (`keyup` & `change`) make immediate changes to other values (without the need for a page refresh or pressing a button):



The actions attached to each of these events call methods from the AppComponent class to set the data members – for example the `setDBName` method (from `app.component.ts`):

```
setDBName(dbName: string) {
  this.dBInputs.MongoDBDatabaseName = dbName;
  this.dBURI = this.dataService.calculateMongoDBURI
    (this.dBInputs);
}
```

In addition to setting the `dBInputs.MongoDBDatabaseName` value, it also invokes the data service method `calculateMongoDBURI` (taken from `data.service.ts`):

```
calculateMongoDBURI(dbInputs: any): {
  "MongoDBURI": string,
  "MongoDBURIRedacted": string}
{
  /*
  Returns the URI for accessing the database; if
  it's for MongoDB Atlas then include the password
  and use the chosen database name rather than
  'admin'. Also returns the redacted URI (with the
  password masked).
  */
  let MongoDBURI: string;
  let MongoDBURIRedacted: string;
  if (dbInputs.MongoDBBaseURI ==
      "mongodb://localhost:27017") {
    MongoDBURI = dbInputs.MongoDBBaseURI
      + "/" + dbInputs.MongoDBDatabaseName
      + "?authSource=admin&socketTimeoutMS="
      + dbInputs.MongoDBSocketTimeout*1000
      + "&maxPoolSize="
      + dbInputs.MongoDBConnectionPoolSize;
    MongoDBURIRedacted = dbInputs.MongoDBBaseURI;
  } else {
    // Can now assume that the URI is in the format
    // provided by MongoDB Atlas
    dbInputs.MongoDBUser =
      dbInputs.MongoDBBaseURI.split('mongodb://')
        [1].split(':')[0];
    MongoDBURI = dbInputs.MongoDBBaseURI
      .replace('<DATABASE>',
        dbInputs.MongoDBDatabaseName)
      .replace('<PASSWORD>',
        dbInputs.MongoDBUserPassword)
      + "&socketTimeoutMS="
      + dbInputs.MongoDBSocketTimeout*1000
      + "&maxPoolSize="
      + dbInputs.MongoDBConnectionPoolSize;
    MongoDBURIRedacted = dbInputs.MongoDBBaseURI
      .replace('<DATABASE>',
        dbInputs.MongoDBDatabaseName)
      .replace('<PASSWORD>', "**********")
      + "&socketTimeoutMS="
      + dbInputs.MongoDBSocketTimeout*1000
      + "&maxPoolSize="
      + dbInputs.MongoDBConnectionPoolSize;
  }

  this.setMongoDBURI(MongoDBURI);
  return({
    "MongoDBURI": MongoDBURI,
    "MongoDBURIRedacted": MongoDBURIRedacted});
}
```

This method is run by the handler associated with any data member that affects the MongoDB URI (base URI, database name, socket timeout, connection pool size, or password). Its purpose is to build a full URI which will then be used for accessing MongoDB; if the URI contains a password then a second form of the URI, `MongoDBURIRedacted` has the password replaced with `**********`.

It starts with a test as to whether the URI has been left to the default `localhost:27017` – in which case it's assumed that there's no need for a username or password (obviously, this shouldn't be used in production). If not, it assumes that the URI has been provided by the MongoDB Atlas GUI and applies these changes:

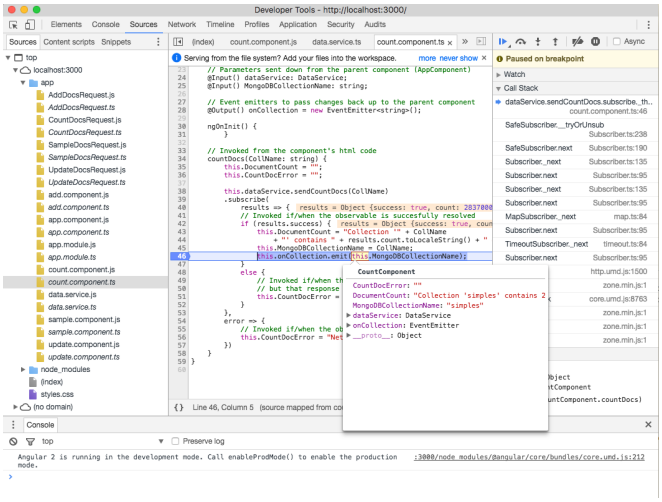
- Change the database name from `<DATATBASE>` to the one chosen by the user.
- Replace `<PASSWORD>` with the real password (and with `**********` for the redacted URI).
- Add the socket timeout parameter.

- Add the connection pool size parameter.

Testing & debugging the Angular application

Now that the full MEAN stack application has been implemented, you can test it from within your browser.

Debugging the Angular 2 client is straightforward using the Google Chrome Developer Tools which are built into the Chrome browser. Despite the browser executing the transpiled JavaScript the Dev Tools allows you to browse and set breakpoints in your Typescript code:



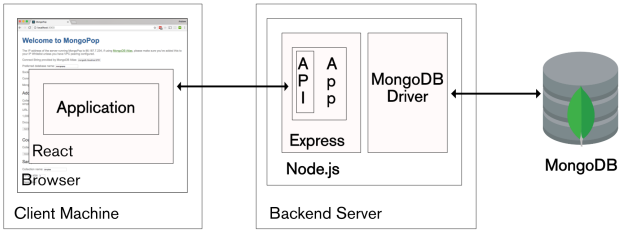
Chapter summary

Previous chapters stepped through building the Mongopop application backend. This chapter describes how to build a frontend client using Angular 2. At this point, we have a complete, working, MEAN stack application.

The coupling between the front and backend is loose; the client simply makes remote, HTTP requests to the backend service.

Using ReactJS, ES6 & JSX to Build a UI (the rise of MERN)

This chapter is similar to the previous one, except that it uses ReactJS rather than Angular to implement a remote web-app client for the **Mongopop** application – completing the full MERN application stack.



React (alternatively referred to as ReactJS), is an up and coming alternative to Angular. It is a JavaScript library, developed by Facebook and Instagram, to build interactive, reactive user interfaces. Like Angular, React breaks the frontend application down into components. Each component can hold its own ***state***and a parent can pass its state down to its child components (as ***properties***) and those components can pass changes back to the parent through the use of callback functions. Components can also include regular data members (which are not state or properties) for data which isn't rendered.

State variables should be updated using the `setState` function – this allows ReactJS to calculate which elements of the page need to be refreshed in order to reflect the change. As refreshing the whole page can be an expensive operation, this can represent a significant efficiency and is a big part of what makes React live up to its name as "reactive".

React components are typically implemented using JSX – an extension of JavaScript that allows HTML syntax to be embedded within the code.

React is most commonly executed within the browser but it can also be run on the backend server within Node.js, or as a mobile app using *React Native*.

JSX & ReactJS

It's possible to implement ReactJS components using 'pure' JavaScript (though, we've already seen that it's more complicated than that) but it's more typical to use JSX. JSX extends the JavaScript syntax to allow HTML and JavaScript expressions to be used in the same code – making the code concise and easy to understand.

Components can be implemented as a single function but we use a class as it offers more options. The following code implements a very simple component:

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
```

By extending `React.Component`, we indicate that the class implements a component and that the `render()` method returns the contents of that component

The enclosing component can pass data down to this component as ***properties*** (accessed within the component as `this.props`); in this case, there is just one – `name`. JavaScript can be included at any point in the returned HTML by surrounding it with braces `{this.props.name}`. The enclosing component would include this code within its own `render()` method, where `userName` is part of that component's state.:

```
<HelloMessage
name={this.state.userName}
/>
```

The ***state*** data member for a component should include all of the variable values that are to be rendered (apart from those that have been passed down as properties). State values can be initialized directly in the class's constructor function but after that, the `setState({userName: "Andrew"})` method should be used so that ReactJS knows that any elements containing `userName` should be re-rendered.

JSX gets compiled into JavaScript before it's used (we use the Babel compiler) and so there are no special dependencies on the browser.

Downloading, running, and using the Mongopop ReactJS application

The compiled ReactJS client code is included as part of the Mongopop package you installed earlier, and the app's backend is started in the same way:

```
git clone \
    git@github.com:am-MongoDB/MongoDB-Mongopop.git
cd MongoDB-Mongopop
npm install
npm run express
```

Run the ReactJS client by browsing to `http://backend-server:3000/react`.

Unlike the Angular client, the ReactJS application is developed and built as a separate project, and then compiled results are copied to `public/react` in the backend server (this is covered in the next section).

Build and deploy

To access the source and build an updated version of the client, a new GitHub repository must be downloaded – MongoDB-Mongopop-ReactJS:

```
git clone git@github.com:am-MongoDB/\
  MongoDB-Mongopop-ReactJS.git
cd MongoDB-Mongopop-ReactJS
```

As with the backend and the Angular client, `package.json` includes a list of dependencies as well as scripts:

```
{
  "name": "mongopop-react-client",
  "version": "0.1.0",
  "private": false,
  "homepage": "http://localhost:3000/react",
  "devDependencies": {
    "react-scripts": "0.8.5"
  },
  "dependencies": {
    "mongodb": "^2.2.20",
    "react": "^15.4.2",
    "react-dom": "^15.4.2"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "eject": "react-scripts eject"
  }
}
```

Before running any of the software, the Node.js dependencies (as defined in `package.json` must be installed into the `node_modules` directory):

```
npm install
```

To compile the JSX code, start the development server, and run the ReactJS client, run:

```
export PORT=3030 # As Express is already using 3000
                 # on this machine
npm start
```

This should automatically open the application within your browser. Note that the ReactJS code was loaded from a

local development server but it will use the real REST API running in the backend.

Note that when running in this mode, you may get errors when your browser tries accessing the REST API – this is because browsers typically block cross-site scripting. To work around this, install this extension from the Google Chrome store.

If you make changes to the ReactJS client and want to include them in the real backend then build a new, optimized version:

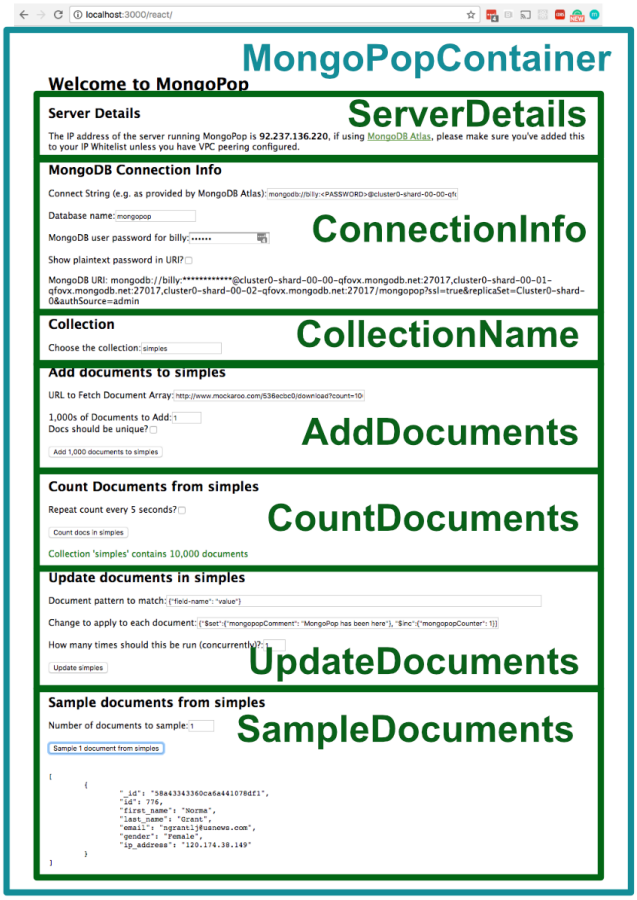
```
npm run build
```

The contents of the `MongoDB-Mongopop-ReactJS/build` folder should then be copied to `MongoDB-Mongopop/public/react`.

To see exactly what `react-scripts` is doing for these operations, review the scripts in `node_modules/react-scripts/scripts`.

Component architecture of the Mongopop ReactJS UI

Most ReactJS applications are built from one or more, nested components – Mongopop is no exception:



The top-level component (`MongoPopContainer`) renders the "Welcome to MongoPop" heading before delegating the the rest of the page to seven sub-components.

`MongoPopContainer` is implemented by a JSX class of the same name. The class contains the state variables for any information which must be used by more than one sub-component (e.g. the collection name). It also includes handler functions that will be used by sub-components when they make changes to any state variable passed down. The class implements the `render()` function which returns the expression that ReactJS must convert to HTML for rendering; in addition to the opening `Welcome to MongoPop` header, it includes an element for each of the sub-components. As part of those element definitions, it passes down state variables (which the sub-component receives as properties):

```
<CountDocuments
  dataService={this.dataService}
  collection={this.state.MongoDBCollectionName}
/>
```

Changes to a data value by a parent component will automatically be propagated to a child – it's best practice to have data flow in this direction as much as possible. If a data value is changed by a child *and* the parent (either directly or as a proxy for one of its other child components) needs to know of the change, then the child triggers an event. That event is processed by a handler registered by the parent – the parent may then explicitly act on the change, but even if it does nothing explicit, the change flows to the other child components.

Each of the sub-components is implemented by its own JSX class – e.g. CountDocuments.

Mongopop is a reasonably flat application with only one layer of sub-components below `MongoPopContainer`, but more complex applications may nest deeper and reuse components.

This table details what data is passed from `MongoPopContainer` down to each of its children and what data change events are sent back up to `MongoPopContainer` (and from there, back down to the other children):

Flow of data between ReactJS components

Child component	Data passed down	Data changes passed back up
`ServerDetails`	Data service	
`ConnectionInfo`	Data service	
`CollectionName`	Data service	Collection Name
`AddDocuments`	Collection Name	
	Data service	
`CountDocuments`	Collection Name	
	Data service	
`UpdateDocuments`	Collection Name	
	Data service	
	Sample data to play with	
`SampleDocuments`	Collection Name	Sample data to play with
	Data service	

What are all of these files?

To recap, the files and folders covered earlier (for the backend, under `MongoDB-Mongopop` folder):

- **`package.json`**: Instructs the Node.js package manager (`npm`) what it needs to do; including which dependency packages should be installed
- **`node_modues`**: Directory where `npm` will install packages
- **`node_modues/mongodb`**: The MongoDB driver for Node.js
- **`node_modues/mongodb-core`**: Low-level MongoDB driver library; available for framework developers (application developers should avoid using it directly)
- **`javascripts/db.js`**: A JavaScript module we've created for use by our Node.js apps (in this book, it will be Express) to access MongoDB; this module in turn uses the MongoDB Node.js driver.
- **`config.js`**: Contains the application–specific configuration options
- **`bin/www`**: The script that starts an Express application; this is invoked by the `npm start` script within the `package.json` file. Starts the HTTP server, pointing it to the `app` module in `app.js`
- **`app.js`**: Defines the main backend application module (`app`). Configures:
  - That the application will be run by Express
  - Which routes there will be & where they are located in the file system (`routes` directory)
  - What view engine to use (Jade in this case)
  - Where to find the *views* to be used by the view engine (`views` directory)
  - What middleware to use (e.g. to parse the JSON received in requests)
  - Where the static files (which can be read by the remote client) are located (`public` directory)
  - Error handler for queries sent to an undefined route
- **`views`**: Directory containing the templates that will be used by the Jade view engine to create the HTML for any pages generated by the Express application (for

this application, this is just the error page that's used in cases such as mistyped routes ("404 Page not found"))

- **routes**: Directory containing one JavaScript file for each Express route
  - **routes/pop.js**: Contains the Express application for the /pop route; this is the implementation of the Mongopop REST API. This defines methods for all of the supported route paths.
- **public**: Contains all of the static files that must be accessible by a remote client (e.g., our Angular to React apps).

In addition, for the ReactJS client application:

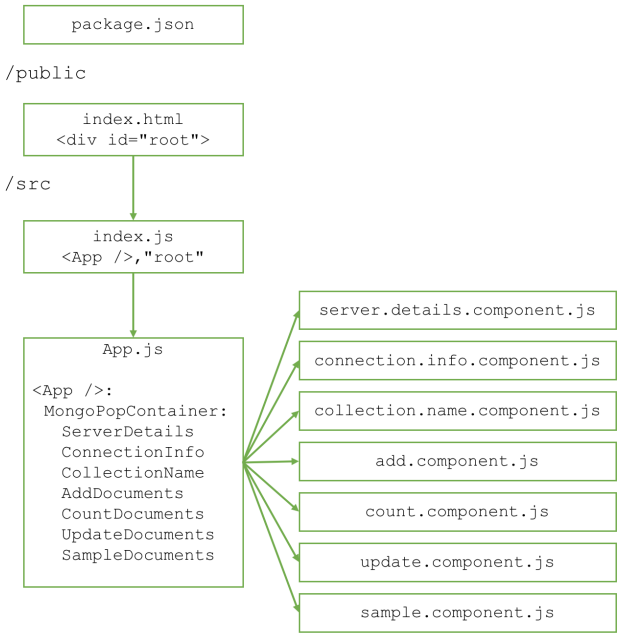
- **public/react** The deployed ReactJS client code; e.g. the JSX code that has been compiled down into vanilla JavaScript

More significant for this chapter are the new files introduced under the MongoDB-Mongopop-ReactJS folder:

- **build**: Directory containing the compiled and optimized JavaScript (to be copied to the backend)
- **node-modules**: Node.js modules used by the ReactJS client application (as opposed to the Express, server-side Node.js modules)
- **public/index.html**: Outer template for the application (includes the rootdiv element)
- **src**: Directory JSX source code files we write for the application
  - **index.js**: Top-level JSX for the client; creates the <App /> element as a placeholder to be expanded by App.js
  - **App.js**: Replaces the <App /> element from index.js with the output from the MongoPopContainer component/class. Includes the rest of the client components
  - **X.component.js**: Class implementing sub-component X
  - **data.service.js**: Service used to interact with the backend REST API (mostly used to access the database)
- **package.json**: Instructs the Node.js package manager (npm) what it needs to do; including which dependency packages should be installed

"Boilerplate" files and how they get invoked

Compared to Angular, you need far fewer source files before reaching the actual application code:



public/index.html defines a div element with its id set to root:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, \
      initial-scale=1">
    <link rel="shortcut icon" href="%PUBLIC_URL%/\
      favicon.ico">
    <title>MongoDB Population Tool - React \
    client</title>
  </head>
  <body>
    <div id="root"></div>
  </body>
</html>
```

src/index.js accesses the root element from public/index.html so that it can be populated with the output from the application. It imports src/App.js and creates the <App /> element.

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```

`src/App.js` defines the `App` class to satisfy the `App` element in `src/index.js`; that class renders the `<MongoPopContainer />` element, which is made up of all of the sub-components. `App.js` imports each of the sub-component source files (`X.component.js`) so that they can implement those components. It also imports `src/data.service.js` to give access to the backend Mongopop REST API.

```
...
class MongoPopContainer extends React.Component {
    ...
}
...
class App extends Component {

  render() {
    return (
      <MongoPopContainer />
      )
  }
}

export default App;
```

Calling the REST API

The Data Service class hides the communication with the backend REST API; serving two purposes:

- Simplifying all of the components' code
- Shielding the components' code from any changes in the REST API signature or behavior – that can all be handled within the DataService

The functions of the data service return promises to make working with their asynchronous behavior simpler.

As a reminder, this is the REST API we have to interact with:

This data access class uses the XMLHttpRequest API to make asynchronous HTTP requests to the REST API running in the backend (mostly to access MongoDB).

One of the simplest functions that `data.service.js` provides is `fetchConfig` which sends an HTTP `GET` request to the backend to retrieve default the client configuration parameters:

```
fetchConfig () {
  /*
  Config data: {
      mongodb: {
          defaultDatabase: string;
          defaultCollection: string;
          defaultUri: string;
      };
      mockarooUrl: string;
  }
  */

  // Ask the MongoPop API for default client
  // config data
  let _this = this;
  return new Promise(function(resolve, reject) {
    var xhr = new XMLHttpRequest();

    xhr.open('GET', _this.baseURL + "/config",
      true);
    xhr.send();
    xhr.addEventListener("readystatechange",
      processRequest, false);
    xhr.onreadystatechange = processRequest;
    xhr.onerror = processError;
    xhr.onabort = processError;

    function processRequest(e) {
      if (xhr.readyState === 4) {
        if (xhr.status === 200) {
          var response = JSON.parse(
            xhr.responseText);
          resolve(response);
        } else {
          var error = xhr.statusText;
          reject("http/app Error: " + error);
        }
      }
    }

    function processError(err) {
        reject("Network Error: " +
          err.target.status);
    }
  })
}
```

When using this API, the application registers handler functions against a number of possible events; in this case:

- `onreadystatechange`: triggered if/when a successful HTTP response is received
- `onerror` & `onabort`: triggered when there has been a problem

The method returns a promise which subsequently – via the bound-in function (`processRequest` & `processError`) – either:

- Provides an object representing the received response
- Raises an error with an appropriate message

The `baseURL` data member is set to `http://localhost:3000/pop` but that can be changed by editing the data service creation line in `App.js`:

```
this.dataService = new DataService(
      "http://localhost:3000/pop");
```

Another of the methods sends a `POST` message to the REST API's `pop/addDocs` route path to request the bulk addition of documents to a MongoDB collection:

```
sendAddDocs(CollName:string, DocURL: string,
            DocCount: number, Unique: boolean) {

  /*
  Use the Mongopop API to add X thousand documents
  (generated) via the supplied Maockaroo URL.
  */

  let _this = this;

  return new Promise(function(resolve, reject) {
    var xhr = new XMLHttpRequest();

    xhr.open('POST', _this.baseURL + "/addDocs",
      true);
    xhr.setRequestHeader("Content-type",
      "application/json;charset=UTF-8");
    xhr.send(JSON.stringify({
      MongoDBURI: _this.MongoDBURI,
      collectionName: CollName,
      dataSource: DocURL,
      numberDocs: DocCount,
      unique: Unique
    }));
    xhr.addEventListener("readystatechange",
      processRequest, false);
    xhr.onreadystatechange = processRequest;
    xhr.onerror = processError;
    xhr.onabort = processError;

    function processRequest(e) {
      let errorText = null;
      if (xhr.readyState === 4) {
        if (xhr.status === 200) {
          var response = JSON.parse(
            xhr.responseText);
          if (response.success) {
            return resolve(response.count);
          } else {
            errorText = "App failure: " +
              response.error;
          }
        } else {
          errorText = "http error: " +
            xhr.statusText;
        }
      }
      if (errorText) {reject(errorText)}
    }

    function processError(err) {
      reject("Network Error: " + err.target.status);
    }
  }
)}
```

The program flow is very similar to that of the previous function and, in the success case, it eventually resolves the returned promise with a count of the number of documents added.

A final method from the `DataService` class worth looking at is `calculateMongoDBURI()` which takes the MongoDB URI provided by MongoDB Atlas and converts it into one that can actually be used to access the database – replacing the `<DATABASE>` and `<PASSWORD>` placeholders with the actual values:

```
calculateMongoDBURI(dbInputs) {
  /*
  Returns the URI for accessing the database; if
  it's for MongoDB Atlas then include the
  password and use the chosen database name
  rather than 'admin'. Also returns the redacted
  URI (with the passwordmasked).

  dbInputs:
  {
      MongoDBBaseURI: string,
      MongoDBDatabaseName: string,
      MongoDBUser: string,
      MongoDBUserPassword: string
  }

  returns:
  {
      MongoDBURI: string,
      MongoDBURIRedacted: string
  }
  */

  let MongoDBURI = "";
  let MongoDBURIRedacted ="";

  if (dbInputs.MongoDBBaseURI ===
      "mongodb://localhost:27017") {
    MongoDBURI = dbInputs.MongoDBBaseURI + "/"
      + dbInputs.MongoDBDatabaseName +
      "?authSource=admin";
    MongoDBURIRedacted = dbInputs.MongoDBBaseURI;
  } else {
    // Can now assume that the URI is in the format
    // provided by MongoDB Atlas
    dbInputs.MongoDBUser = dbInputs.MongoDBBaseURI.
      split('mongodb://')[1].split(':')[0];
    MongoDBURI = dbInputs.MongoDBBaseURI
      .replace('<DATABASE>',
        dbInputs.MongoDBDatabaseName)
      .replace('<PASSWORD>',
        dbInputs.MongoDBUserPassword);
    MongoDBURIRedacted = dbInputs.MongoDBBaseURI
      .replace('<DATABASE>',
        dbInputs.MongoDBDatabaseName)
      .replace('<PASSWORD>',
        "************");
  }

  this.MongoDBURI = MongoDBURI;
  return({"MongoDBURI": MongoDBURI,
    "MongoDBURIRedacted": MongoDBURIRedacted});
}
```

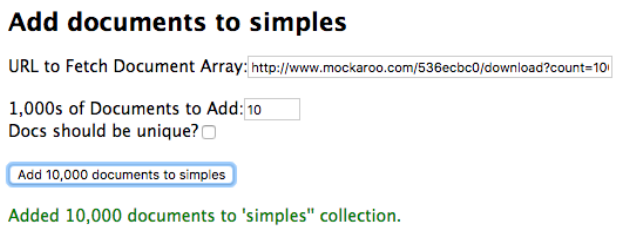
The function stores the final URI in the data service class's `MongoDBURI` data member – to sent to the backend when accessing the database (see `sendAddDocs` above). It also returns a second value (`MongoDBURIRedacted`) with the password masked out – to be used when displaying the URI.

A simple component that accepts data from its parent

Recall that the application consists of eight components: the top-level application which contains each of the `ServerDetails`, `ConnectionInfo`, `CollectionName`, `AddDocuments`, `CountDocuments`, `UpdateDocuments`, and `SampleDocuments` components.

When building a new application, you would typically start by designing the the top-level component and then working downwards. As the top-level container is, perhaps, the most complex one to understand, we'll start at the bottom and then work up.

A simple sub-component to start with is the `AddDocuments` component:

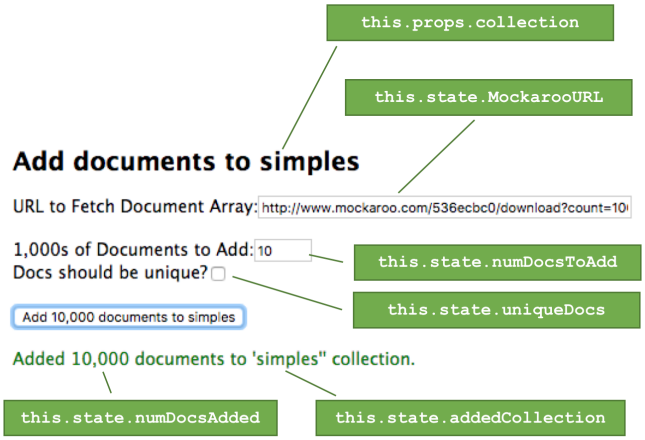


A central design decision for any component is what ***state*** is required (any variable data that is to be rendered by the component should either be part of the component's state or of the `properties` passed by its parent component. The state is initialized in the class's constructor:

```
export class AddDocuments extends React.Component {

  constructor(props) {
    super(props);
    this.state = {
      MockarooURL: "",
      numDocsToAdd: 1,
      uniqueDocs: false,
      numDocsAdded: "",
      errorText: "",
      addedCollection: this.props.collection
    }

    this.handleURLChange=
      this.handleURLChange.bind(this);
    this.handleNumDocsToAddChange=
      this.handleNumDocsToAddChange.bind(this);
    this.handleUniqueToggle=
      this.handleUniqueToggle.bind(this);
    this.handleAddSubmit=
      this.handleAddSubmit.bind(this);
  }
...
}
```

Recall that any state variable X can be read using `this.state.X` but only the constructor should write to it that way – anywhere else should use the `setState()` function so that ReactJS is made aware of the change – enabling it to refresh any affected elements. In this class, there are six state variables:

- `MockarooURL`: The URL from a service such as Mockaroo which will return an array containing a set of example JSON documents
- `numDocsToAdd`: How many batches of documents should be added (with the default value of `MockarooURL`, each batch contains 1,000 documents)
- `uniqueDocs`: Whether each batch should be distinct from the other batches (this significantly slows things down)
- `numDocsAdded`: Updated with the number of added documents in the event that the operation succeeds
- `errorText`: Updated with an error message in the event that the operation fails
- `addedCollection`: Name of the collection that documents were last added to (initialized with the `collection` property passed by the parent component)



Note that the constructor receives the properties passed down from the parent component. The constructor from the `React.Component` class must always be invoked within the component's constructor: `super(props);`.

The `binds` at the end of the constructor make `this` available for use within the class's methods.

Further down in the class is the `render()` method which returns the content that ReactJS converts to HTML and JavaScript for the browser to render:

```
render() {
  return (
    <div>
      <h2>Add documents to {this.props.collection}
      </h2>
      <div>
        <label>
          URL to Fetch Document Array:
          <input type="text" size="50"
            value={this.state.MockarooURL}
            onChange={this.handleURLChange}
          />
        </label>
        <br/><br/>
        <label>
          1,000s of Documents to Add:
          <input type="number" min="1" max="1000"
            value={this.state.numDocsToAdd}
            onChange={
              this.handleNumDocsToAddChange}
          />
        </label>
        <br/>
        <label>
          Docs should be unique?
          <input type="checkbox"
            checked={this.state.uniqueDocs}
            onChange={this.handleUniqueToggle}
          />
        </label>
        <br/>
        {(this.state.uniqueDocs) ? <span
          className="warningMessage">WARNING:
          Requiring all documents to be unique will
          slow things down</span> : ""
        }
        <br/>
        <button onClick={this.handleAddSubmit}>
          {"Add " + this.state.numDocsToAdd
                  + ",000 documents to "
                  + this.props.collection}
        </button>
        <br/><br/>
        <span className="successMessage">
          {(this.state.numDocsAdded) ? ("Added "
            + this.state.numDocsAdded.
                toLocaleString()
            + " documents to '"
            + this.state.addedCollection
            + "' collection.") : ""}
        </span>
        <span className="errorMessage">
          {(this.state.errorText) ?
            this.state.errorText : ""}
        </span>
      </div>
    </div>
    );
  }
```

Recall that when coding in JSX, JavaScript can be embedded in the HTML by surrounding it with braces. The function uses that almost immediately to include the collection name in the component's header: `<h2>Add documents to {this.props.collection}</h2>`.

The first `input` is initialized with `this.state.MockarooURL` and if the user changes the

value then `this.handleURLChange` is invoked – which in turn updates the state value:

```
handleURLChange(event) {
    this.setState({MockarooURL:
      event.target.value});
}
```

The same pattern holds for the `inputs` for `numDocsToAdd` & `uniqueDocs`.

When this component's button is pressed, the `onClick` event calls `this.handleAddSubmit()`:

```
handleAddSubmit(event) {
  let _this = this;
  this.setState({
    numDocsAdded: null,
    erorText: null,
    addedCollection: this.props.collection});

  this.props.dataService.sendAddDocs(
        this.props.collection,
        this.state.MockarooURL,
        this.state.numDocsToAdd,
        this.state.uniqueDocs)
  .then (
    function(results) {
      _this.setState({numDocsAdded:
        results * 1000});
    },
    function(err) {
      _this.setState({errorText: err});
  })
}
```

This function invokes the `sendAddDocs()` method of the data service that was passed down from the parent component (and so is part of `this.props`). `sendAddDocs()` returns a promise and the first function in the `then` clause is called if/when that promise is successfully resolved – setting the `numDocsAdded` state to the number of added documents; if the promise is instead rejected then the second function is called – setting the error message. In either case, the state change will cause the associated element to be re-rendered:

```
<span className="successMessage">
  {(this.state.numDocsAdded) ? ("Added " +
      this.state.numDocsAdded.toLocaleString() +
      " documents to '" +
      this.state.addedCollection + "' collection.")
      : ""}
</span>
<span className="errorMessage">
  {(this.state.errorText) ?
    this.state.errorText : ""}
</span>
```

Passing data down to a sub-component (and receiving changes back)

The `AddDocs` component is embedded within the `render()` method of `MongoPopContainer` component class; implemented in `App.js`:

```
class MongoPopContainer extends React.Component {
    ...
  render() {

    return (
      <div>
      <h1>Welcome to MongoPop</h1>
          ...
        <AddDocuments
          dataService={this.dataService}
          collection={
            this.state.MongoDBCollectionName}
        />
          ...
    );
  }
}
```

It passes down two items:

- `dataService` is an instance of the `DataService` class and is used to access the backend (in particular, to interact with MongoDB). Appears as part of `AddDocument`'s properties and can be accessed as `this.props.dataService`.
- `collection` is a string representing the collection name. Appears as part of `AddDocument`'s properties and can be accessed as `this.props.collection`.

`MongoDBCollectionName` is initialized, and `dataService` is instantiated as part of the `MongoPopContainer` constructor:

```
class MongoPopContainer extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      MongoDBCollectionName: "",
      DataToPlayWith: false
    };

    this.dataService =
      new DataService("http://localhost:3000/pop");

    this.handleCollectionChange =
      this.handleCollectionChange.bind(this);
    this.handleDataAvailabiltyChange =
      this.handleDataAvailabiltyChange.bind(this);
}
```

Note that for a real, deployed application, `http://localhost:3000/pop` would be replaced with

the public URL for REST API. Additionally, you should consider adding authentication to the API .

But where did the collection name get set – the constructor initialized it to an empty string but that's not we see when running the application? There's a clue in the constructor:

```
this.handleCollectionChange =
  this.handleCollectionChange.bind(this);
```

Recall that a `bind` like this is to allow a function (`this.handleCollectionChange()`) to access the `this` object:

```
handleCollectionChange(collection) {
    this.setState({MongoDBCollectionName:
      collection});
}
```

The `handleCollectionChange()` method is passed down to the `CollectionName` component:

```
class MongoPopContainer extends React.Component {
    ...
  render() {

    return (
        ...
      <CollectionName
        dataService={this.dataService}
        onChange={this.handleCollectionChange}
      />
        ...
    );
  }
}
```

This is the `CollectionName` component class:

```
import React from 'react';
import './App.css';

export class CollectionName extends React.Component
{

  constructor(props) {
    super(props);

    this.state = {
      collection: ""
    }

    this.handleCollectionNameChange =
      this.handleCollectionNameChange.bind(this);
  }

  componentDidMount() {

  /* Fetch default client config information from
  the backend. Expect to receive:

      {
          mongodb: {
            defaultDatabase: string;
            defaultCollection: string;
            defaultUri: string;
          };
          mockarooUrl: string;
        }
    */

    let _this = this;

    this.props.dataService.fetchConfig ()
    .then(
      function(results) {
      _this.setState({collection:
        results.mongodb.defaultCollection},
        () => {
        _this.props.onChange(
          _this.state.collection);});
      },
      function(err) {
        console.log ("fetchConfig: Hit problem: " +
          err);
      }
    )
  }

  handleCollectionNameChange(event) {
    this.setState({collection:
      event.target.value});
    this.props.onChange(event.target.value);
  }

  render() {
    return (
      <div>
        <h2>Collection</h2>
        <label>
          Choose the collection:
          <input type="text" size="20"
            value={this.state.collection}
            onChange={
              this.handleCollectionNameChange}
          />
        </label>
      </div>
    );
  }
}
```

`CollectionName` has a single state variable – `collection` – which is initially set in the `componentDidMount()` method by fetching the default client configuration information from the backend by calling `this.props.dataService.fetchConfig()`. `componentDidMount` is one of the component lifecycle methods that are part of any `React.Component` class – it is invoked after the component has been loaded into the browser, it is where you would typically fetch any data from the backend that's needed for the component's starting state. After setting the collection state, the change notification function passed down by the parent component is invoked to pass up the new value:

```
_this.props.onChange(_this.state.collection);
```

Of course, the user needs to be able to change the collection name and so an `input` element is included. The value of the element is initialized with the collection state variable and when the user changes that value, `this.handleCollectionNameChange` is invoked. In turn, that method updates the component state and passes the new collection name up to the parent component by calling the change notification method provided by the parent.

Optionally empty components

It's common that a component should only display its contents if a particular condition is met. Mongopop includes a feature to allow the user to apply a bulk change to a set of documents – selected using a pattern specified by the user. If they don't know the typical document structure for the collection then it's unlikely that they'll make a sensible change. Mongopop forces them to first retrieve a sample of the documents before they're given the option to make any changes.

This optionality is implemented through the `SampleDocuments` & `UpdateDocuments` components:

Flow of data between ReactJS components

Child component	Data passed down	Data changes passed back up
`UpdateDocuments`	Collection Name	
	Data service	
	Sample data to play with	
`SampleDocuments`	Collection Name	Sample data to play with
	Data service	

Recall that the `MongoPopContainer` component class includes a state variable named `DataToPlayWith` which is initialized to `FALSE`:

```
class MongoPopContainer extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      MongoDBCollectionName: "",
      DataToPlayWith: false
    };

    this.dataService =
      new DataService("http://localhost:3000/pop");

    this.handleCollectionChange =
      this.handleCollectionChange.bind(this);
    this.handleDataAvailabiltyChange =
      this.handleDataAvailabiltyChange.bind(this);
}
```

That state is updated using the `handleDataAvailabiltyChange` method:

```
handleDataAvailabiltyChange(dataAvailable) {
    this.setState({DataToPlayWith: dataAvailable});
}
```

That method is passed down to the `SampleDocuments` component:

```
<SampleDocuments
  dataService={this.dataService}
  collection={this.state.MongoDBCollectionName}
  onDataToWorkWith={
    this.handleDataAvailabiltyChange}
/>
```

When the user fetches a sample of the documents from a collection, the `SampleDocuments` component invokes the change notification method (`_this.props.onDataToWorkWith()`), passing back `TRUE` if the request was a success, `FALSE` otherwise:

```
import React from 'react';
import './App.css';

export class SampleDocuments
  extends React.Component {

  constructor(props) {
    super(props);
    this.state = {
      numDocsToSample: 1,
      documents: null,
      errorText: null
    }

    this.handleNumDocsToSampleChange =
      this.handleNumDocsToSampleChange.bind(this);
    this.handleSampleSubmit =
      this.handleSampleSubmit.bind(this);
  }

  componentDidMount() {
  }

  handleNumDocsToSampleChange(event) {
    this.setState({numDocsToSample:
      event.target.value})
  }

  handleSampleSubmit(event) {
    let _this = this;

    this.setState({
      documents: null,
      erorText:  null});

    this.props.dataService.sendSampleDocs(
      this.props.collection,
      this.state.numDocsToSample)
    .then (
      function(results) {
        _this.setState({documents: JSON.stringify(
          results, null, `\t`)});
        _this.props.onDataToWorkWith(true);
      },
      function(err) {
        _this.setState({errorText: err});
        _this.props.onDataToWorkWith(false);
      })
  }

  render() {
    return (
      <div>
        <h2>Sample documents from
          {this.props.collection}</h2>
        <div>
          <label>
            Number of documents to sample:

            <input type="number" min="1" max="100"
              value={this.state.numDocsToSample}
              onChange={
                this.handleNumDocsToSampleChange}
            />
          </label>
          <br/><br/>
          <button onClick={this.handleSampleSubmit}
          > {"Sample " + this.state.numDocsToSample
            + (" document from " :
              " documents from ")
            + this.props.collection}
          </button>
          <br/><br/>
          <span className="json">
          <pre>{(this.state.documents) ?
            this.state.documents : ""}</pre>
          </span>
          <span className="errorMessage">
            {(this.state.errorText) ?
              this.state.errorText : ""}
          </span>
        </div>
      </div>
    );
  }
}
```

`MongoPopContainer` passes its state variable `DataToPlayWith` down to the `UpdateDocuments` component:

```
<UpdateDocuments
  dataService={this.dataService}
    collection={this.state.MongoDBCollectionName}
  dataToPlayWith={this.state.DataToPlayWith}
/>
```

The `UpdateDocuments` component class is then able to check the value using:

```
render() {
  return (
    this.props.dataToPlayWith ? (
      // code for the component to use
    ) : null
  );
}
```

Otherwise, the rest of this component is similar to those already seen:

```
import React from 'react';
import './App.css';

export class UpdateDocuments
  extends React.Component {

  constructor(props) {
    super(props);
    this.state = {
      matchPattern: '{"field-name": "value"}',
      changePattern: '{"$set":{"mongopopComment":
          "MongoPop has been here"},
        "$inc":{"mongopopCounter": 1}}',
      threads: 1,
      numDocsUpdated: null,
      errorText: null,
      updatedCollection: this.props.collection
    }
    this.handleMatchPatternChange =
      this.handleMatchPatternChange.bind(this);
    this.handleChangePatternChange =
      this.handleChangePatternChange.bind(this);
    this.handleThreadsChange =
      this.handleThreadsChange.bind(this);
    this.handleUpdateSubmit =
      this.handleUpdateSubmit.bind(this);
  }

  componentDidMount() {
  }

  handleMatchPatternChange(event) {
    this.setState({
      matchPattern: event.target.value});
  }

  handleChangePatternChange(event) {
    this.setState(
      {changePattern: event.target.value})
  }

  handleThreadsChange(event) {
    this.setState({threads: event.target.value})
  }

  handleUpdateSubmit(event) {
    let _this = this;
    this.setState({
      numDocsUpdated: null,
      errorText: null,
      updatedCollection: this.props.collection});

    this.props.dataService.sendUpdateDocs(
      this.props.collection,
      this.state.matchPattern,
      this.state.changePattern,
      this.state.threads)
    .then (
      function(results) {
        _this.setState({numDocsUpdated: results});
      },
      function(err) {
        _this.setState({errorText: err});
      })
  }

  render() {
    return (
      this.props.dataToPlayWith ? (
        <div>
          <h2>Update documents in
            {this.props.collection}</h2>
          <div>
            <label>
              Document pattern to match:
              <input type="text" size="100"
                value={this.state.matchPattern}
                onChange={
                  this.handleMatchPatternChange}
              />
            </label>
            <br/><br/> <label>
              Change to apply to each document:
              <input type="text" size="80"
                value={this.state.changePattern}
                onChange={
                  this.handleChangePatternChange}
              />
            </label>
            <br/><br/>
            <label>
              How many times should this be run
                (concurrently)?:
              <input type="number" min="1" max="80"
                value={this.state.threads}
                onChange={this.handleThreadsChange}
              />
            </label>
            <br/>
            <br/>
            <button onClick={
              this.handleUpdateSubmit}>
              {"Update " + this.props.collection}
            </button>
            <br/><br/>
            <span className="successMessage">
              {(this.state.numDocsUpdated != null)
                ? ("Updated " +
                this.state.numDocsUpdated
                  .toLocaleString() +
                " documents in the '" +
                this.state.updatedCollection +
                "' collection.") : ""}
            </span>
            <span className="errorMessage">
              {(this.state.errorText) ?
                this.state.errorText : ""}
            </span>
          </div>
        </div>
      ) : null
    );
  }
}
```

Periodic operations

The `CountDocuments` component has an extra feature – if the repeat option is checked then it will fetch and display the document count every five seconds. The function that's called when the count button is clicked, checks the value of the state variable associated with the checkbox and if it's

set, calls `setInterval()` to call the `countOnce()` method every five seconds:

```
if (this.state.repeat) {
  this.timerID = setInterval(
    () => this.countOnce(),
    5000
  );
}
```

The timer is cleared (`clearInterval()`) if there is an error or just before the component is unmounted (in `componentWillUnmount`).

Other components

For completeness, this is the full top-level component, `App.js`, which includes the rest of the sub-components:

```
import React, {Component} from 'react';
import './App.css';
import {DataService} from './data.service';
import {ServerDetails} from
  './server.details.component';
import {ConnectionInfo} from
  './connection.info.component';
import {CollectionName} from
  './collection.name.component';
import {CountDocuments} from './count.component';
import {AddDocuments} from './add.component';
import {UpdateDocuments} from './update.component';
import {SampleDocuments} from './sample.component';

class MongoPopContainer extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      MongoDBCollectionName: "",
      DataToPlayWith: false
    };

    this.dataService = new DataService(
      "http://localhost:3000/pop");

    this.handleCollectionChange =
      this.handleCollectionChange.bind(this);
    this.handleDataAvailabiltyChange =
      this.handleDataAvailabiltyChange.bind(this);

  }

  componentDidMount() {
  }

  handleCollectionChange(collection) {
    this.setState({MongoDBCollectionName:
      collection});
  }

  handleDataAvailabiltyChange(dataAvailable) {
    this.setState({DataToPlayWith:
      dataAvailable});
  }

  render() {

    return (
      <div>
      <h1>Welcome to MongoPop</h1>
        <ServerDetails
          dataService={this.dataService}
        />
        <ConnectionInfo
          dataService={this.dataService}
        />
        <CollectionName
          dataService={this.dataService}
          onChange={this.handleCollectionChange}
        />
        <AddDocuments
          dataService={this.dataService}
          collection={
            this.state.MongoDBCollectionName}
        />
        <CountDocuments
          dataService={this.dataService}
          collection={
            this.state.MongoDBCollectionName}
        />
        <UpdateDocuments
          dataService={this.dataService}
          collection={
            this.state.MongoDBCollectionName}
          dataToPlayWith={
            this.state.DataToPlayWith}
        />
        <SampleDocuments
          dataService={this.dataService}
          collection={
            this.state.MongoDBCollectionName}
          onDataToWorkWith={
            this.handleDataAvailabiltyChange}
        />
      </div>
    );
  }
}

class App extends Component {

  render() {
    return (
      <MongoPopContainer />
      )
  }
}

export default App;
```

The `ConnectionInfo` component:

```
import React from 'react';
import './App.css';

export class ConnectionInfo extends React.Component
{
  constructor(props) {
    super(props);
    this.state = {
      MongoDBBaseURI: "",
      MongoDBDatabaseName: "",
      MongoDBUser: "",
      needCredentials: false,
      showPassword: false,
      MongoDBURI: "",
      MongoDBURIRedacted: ""
    }

    this.MongoDBUserPassword = "";

    //this = this.props.connectionData;

    this.componentDidMount =
    this.componentDidMount.bind(this);
    this.handleBaseURIChange =
    this.handleBaseURIChange.bind(this);
    this.handleDatabaseNameChange =
    this.handleDatabaseNameChange.bind(this);
    this.handlePasswordChange =
    this.handlePasswordChange.bind(this);
    this.handlePasswordToggle =
    this.handlePasswordToggle.bind(this);
  }

  componentDidMount() {

    /* Fetch default client config information from
    the backend. Expect to receive:

      {
          mongodb: {
              defaultDatabase: string;
              defaultCollection: string;
              defaultUri: string;
          };
          mockarooUrl: string;
        }
        */

    let _this = this;

    this.props.dataService.fetchConfig ()
    .then(
      function(results) {
        _this.setState({MongoDBBaseURI:
          results.mongodb.defaultUri},
        () => {
          _this.setState({MongoDBDatabaseName:
          results.mongodb.defaultDatabase},
        () => {
          _this.handleConnectionChange();
        });
      });
    },
    function(err) {
      console.log ("fetchConfig: Hit problem: " +
        err);
    }
  )
}

handleBaseURIChange(event) {
  this.setState({MongoDBBaseURI:
      event.target.value},
      () => {
        this.setState({needCredentials: true},
        () => {
      this.handleConnectionChange();
    });
});
}

handleDatabaseNameChange(event) {
  this.setState({MongoDBDatabaseName:
      event.target.value},
    () => {
      this.handleConnectionChange();
    });
}

handlePasswordChange(event) {
  this.MongoDBUserPassword = event.target.value;
  this.handleConnectionChange();
}

handlePasswordToggle(event) {
  this.setState({showPassword:
    event.target.checked});
}

handleConnectionChange() {

  var dBInputs = {
    MongoDBBaseURI: this.state.MongoDBBaseURI,
    MongoDBDatabaseName:
      this.state.MongoDBDatabaseName,
    MongoDBUser: this.state.MongoDBUser,
    MongoDBUserPassword: this.MongoDBUserPassword
  }

  const dBURI = this.props.dataService
    .calculateMongoDBURI(dBInputs);
  this.setState({MongoDBURI: dBURI.MongoDBURI});
  this.setState({MongoDBURIRedacted:
    dBURI.MongoDBURIRedacted});
```

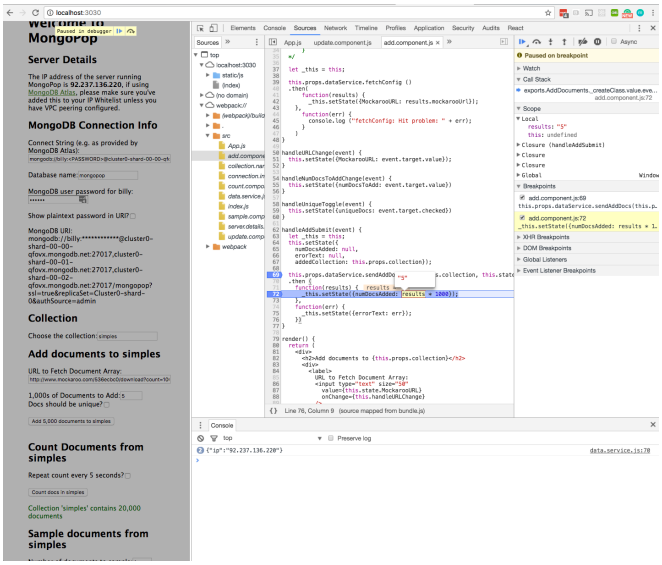
The `ServerDetails` component:

```
import React from 'react';

export class ServerDetails extends React.Component
{
  constructor(props) {
    super(props);

    this.state = {serverIP: ""};

    this.componentDidMount =
      this.componentDidMount.bind(this);
  }

  componentDidMount() {
    let _this = this;

    this.props.dataService.fetchServerIP ()
    .then(
        function(results) {
          _this.setState({serverIP: results});
      },
        function(err) {
          _this.setState({serverIP: "Hit problem: "
            + err});
        }
      )
  }

  render() {
    return(
      <div>
        <h2>Server Details</h2>
        <p>The IP address of the server running
        MongoPop is <strong>{this.state.serverIP}
        </strong>, if using
        <a href="https://cloud.mongodb.com/"
          name="MongoDB Atlas - MongoDB as a \
            service"
          target="_blank">MongoDB Atlas</a>,
        please make sure you've added this to your
        IP Whitelist unless you have VPC peering
        configured.</p>
      </div>
    )
  }
}
```

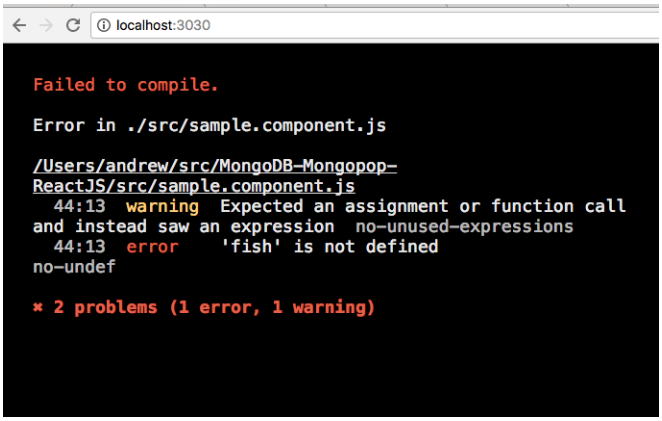
Testing & debugging the ReactJS application

Now that the full MERN stack application has been implemented, you can test it from within your browser.

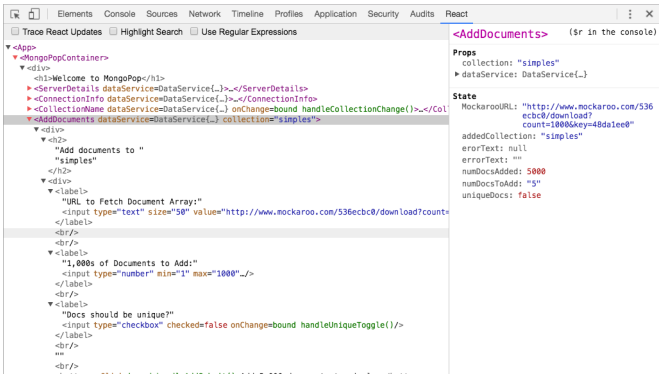
Debugging the ReactJS client is straightforward using the Google Chrome Developer Tools which are built into the Chrome browser. Despite the browser executing the transpiled JavaScript the Dev Tools allows you to navigate and set breakpoints in your JSX code:



If there is a compilation error then the error is sent to the browser:



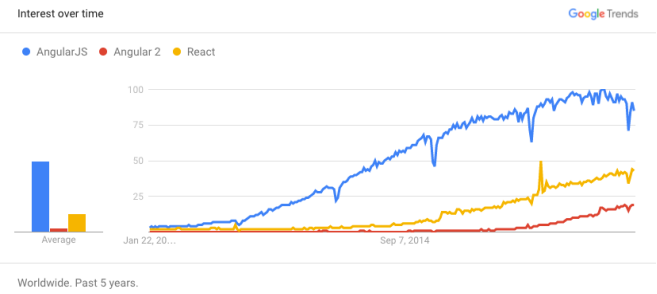
By installing the *React Developer Tools* from the Google Chrome Store, you get an extra "React" tab that can be used to view or modify the state or properties for any of the components:



ReactJS vs. Angular

So should you use Angular 2 or React for your new web application? A quick Google search will find you some fairly deep comparisons of the two technologies but in summary, Angular 2 is a little more powerful while React is easier for developers to get up to speed with and use (note how many fewer files are needed). The previous chapter built the Mongopop client application using Angular 2, while this one built a near-identical app using ReactJS – hopefully these examples have helped you pick a favorite.

The snapshot from Google Trends suggests that Angular has been much more common for a number of years but that React is gaining ground.



Chapter summary

This chapter described how to build a frontend client using ReactJS. At this point, we have a complete, working, MERN stack application.

The coupling between the front and backend is loose; the client simply makes remote, HTTP requests to the backend service.

Browsers Aren't the Only UI – Mobile Apps, Amazon Alexa, Cloud Services...

Once your application backend exposes a REST API, there are limitless ways that you, or other developers can access it:

- A dedicated browser-based client
- A standalone native iOS or Android mobile app
- Voice controlled appliances, such as Amazon's Echo
- IoT-enabled devices, such as remote sensors
- Integrations with 3rd party applications

This chapter takes a look at some of these approaches. Unlike some of the earlier chapters, this one aims to go wide rather than deep – touching on many technologies rather than diving too deeply into any one.

Prerequisite – the REST API

Everything that follows assumes that you have the *Mongopop REST API* running. Additionally, that API has been extended with 3 new routes (already included in the latest GitHub repository):

Additional Express routes implemented for the Mongopop REST API

Route	HTTP Method	Parameters	Response	Purpose
`/pop/checkIn`	POST	`{ venue: string, date: string, url: string, location: string }`	`{ success: boolean, error: string }`	Stores the checkin data as a document in a collection.
`/pop/checkInCount`	GET		`{ success: boolean, count: number, error: string }`	Returns a count for the total number of checkins.
`/pop/latestCheckIn`	GET		`{ success: boolean, venue: string, date: string, url: string, location: string, error: string }`	Retrieves the most recent checkin.

These route paths are implemented in the `pop.js` module in the Mongopop repository:

```
router.post('/checkIn', function(req, res, next) {

  /* Request from client to add a sample of the
  documents from a collection; the request
  should be of the form:

  {
      venue,
      date,
      url,
      location
  }

  The response will contain:

  {
      success: boolean,
      error: string
  }
  */

  var requestBody = req.body;
  var database = new DB;

  database.connect(config.makerMongoDBURI)
  .then(
    function() {

      var checkIn = {
        venueName: requestBody.venue,
        date: requestBody.date,
        url: requestBody.url,
        mapRef: requestBody.location
      }

      // Returning will pass the promise returned
      // by addDoc to the next .then in the chain
      return database.addDocument(
        config.checkinCollection, checkIn)
    })  // No function is provided to handle the
        // connection failing and so that error
        // will flow through to the next .then
  .then(
    function(docs) {
      return {
      "success": true,
      "error": ""
    };
    },
    function(error) {
      console.log('Failed to add document: ' +
        error);
      return {
        "success": false,
        "error": "Failed to add document: " + error
      };
    })
  .then(
    function(resultObject) {
        database.close();
        res.json(resultObject);
    }
  )
})

router.get('/checkInCount',
  function(req, res, next) {

  /* Request from client for the number of checkins

  The response will contain:

  {
      success: boolean,
      count: number,
      error: string
  }
  */

  var requestBody = req.body;
  var database = new DB;

  database.connect(config.makerMongoDBURI)
  .then(
    function() {

      // Returning will pass the promise returned
      // by countDocuments to the next .then in the
      // chain
      return database.countDocuments(
        config.checkinCollection)
    })  // No function is provided to handle the
        // connection failing and so that
        // error will flow through to the next
        // .then
  .then(
    function(count) {
      return {
        "success": true,
        "count": count,
        "error": ""
      };
    },
    function(error) {
      console.log('Failed to count checkins: ' +
        error);
      return {
        "success": false,
        "count": 0,
        "error": "Failed to count checkins: " +
        error
      };
    })
  .then(
    function(resultObject) {
      database.close();
      res.json(resultObject);
    }
  )
})

router.get('/latestCheckIn',
  function(req, res, next) {

  /* Request from client for the number of checkins

  The response will contain:

  {
      success: boolean,
      venue,
      date,
      url,
      location
      error: string
  }
  */
```

```
  var requestBody = req.body;
  var database = new DB;

  database.connect(config.makerMongoDBURI)
  .then(
    function() {

      // Returning will pass the promise returned
      // by mostRecentDocument to the next .then in
      // the chain
      return database.mostRecentDocument(
        config.checkinCollection)
    })  // No function is provided to handle the
        // connection failing and so that error
        // will flow through to the next .then
  .then(
    function(doc) {
      return {
        "success": true,
        "venue": doc.venueName,
        "date": doc.date,
        "url": doc.url,
        "location": doc.mapRef,
        "error": ""
      };
    },
    function(error) {
      console.log('Failed to find last checkin: ' +
        error);
      return {
        "success": false,
        "venue": "",
        "date": "",
        "url": "",
        "location": "",
        "error": "Failed to find last checkin: " +
        error
      };
    })
  .then(
    function(resultObject) {
      database.close();
      res.json(resultObject);
    }
  )
}
```

`/pop/lastCheckIn` depends on a new method that has been added to `javascripts/db.js`:

```
DB.prototype.mostRecentDocument = function(coll) {

  // Return a promise that either resolves with
  // the most recent document from the collection
  // (based on a reverse sort on `_id` or is
  // rejected with the error received from the
  // database.

  var _this=this;

  return new Promise(function (resolve, reject) {
    _this.db.collection(coll, {strict:false},
      function(error, collection){
      if (error) {
        console.log("Could not access collection: "
          + error.message);
        reject(error.message);
      } else {
        var cursor = collection.find({})
          .sort({_id: -1}).limit(1);
        cursor.toArray(function(error, docArray) {
          if (error) {
            console.log(
              "Error reading fron cursor: " +
              error.message);
            reject(error.message);
          } else {
            resolve(docArray[0]);
          }
        })
      }
    })
  })
}
```

The configuration file `config.js` is also extended – note that you should replace the value associated with the `makerMongoDBURI` field if you're not running MongoDB on your local machine (e.g. with the URI provided by MongoDB Atlas):

```
var config = {
  expressPort: 3000,
  client: {
    mongodb: {
      defaultDatabase: "mongopop",
      defaultCollection: "simples",
      defaultUri: "mongodb://localhost:27017"
    },
    mockarooUrl: "http://www.mockaroo.com/536ecbc0/\
      download?count=1000&key=48da1ee0"
  },
  makerMongoDBURI: "mongodb://localhost:27017/\
    maker?authSource=admin",
  checkinCollection: "foursq"
};

module.exports = config;
```

The implementation of these methods follows the same pattern as already seen and so they're not explained here.

Repurposing Angular & ReactJS code for native applications

There are frameworks for both Angular and ReactJS that enable web client application designs (and in some cases, code) to be reused for creating *native* iOS and Android apps.

One option for Angular is **NativeScript**, in which you use Typescript/JavaScript with Angular to build native apps for multiple platforms from the same source code. Of course, to get the most out of those platforms, you may want or need to add platform-specific code.

React developers will find **React Native** code very familiar, and applications are built from declarative components in the same way. The most obvious difference is that React Native code uses its own native components (e.g. `<View>` and `<Text>` rather that HTML elements such as `<div>` and `<p>`):

```
import React, { Component } from 'react';
import { AppRegistry, Text } from 'react-native';

class HelloWorldApp extends Component {
  render() {
    return (
        <View>
          <Text>Hello There!</Text>
          <Text>This is a very simple Native React
           component.</Text>
        </View>
    );
  }
}

AppRegistry.registerComponent('HelloWorldApp',
  () => HelloWorldApp);
```

React Native provides the Fetch API to make network requests; it follows a similar patter to `XMLHttpRequest` (React Native also includes `XMLHttpRequest` which can be used directly).

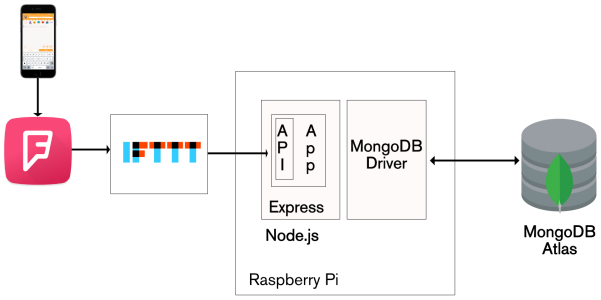
While it's not as simple as just rebuilding your ReactJS or Angular code to produce native apps, the reuse of designs, skills and (some) code make it much more efficient than starting from scratch.

Combining cloud services – IFTTT

IFTTT (IF This Then That) is a free cloud service which allows you to automate tasks by combining existing services (Google Docs, Facebook, Instagram, Hue lights, Nest thermostats, GitHub, Trello, Dropbox,...). The name of the service comes from the simple pattern used for each Applet (automation rule): "**IF T**his event occurs in service x **T**hen trigger **T**hat action in service y".

IFTTT includes a Maker service which can handle web requests (triggers) or send web requests (actions). In this case, I use it to invoke the `pop/checkIn` POST method from the Mongopop REST API whenever I check in using the Swarm (Foursquare) app:

Note that the applet makes a POST request to the `http://your-mongopop-ip:3000/pop/checkIn` route. The body of the POST includes the required parameters – provided as a JSON document. Each of the `VenueName`, `CheckinDate`, `VenueUrl`, and `VenueMapImageURL` values are /ingredients/ from the trigger (Foursquare) event.



Using the Swarm app I check into FourSquare:

We can confirm that the MongoDB collection has been updated after this check-in:

```
Cluster0-shard-0:PRIMARY> use maker
switched to db maker

Cluster0-shard-0:PRIMARY> db.foursq.find()
  .sort({_id: -1}).limit(1).pretty()
{
    "_id" : ObjectId("598daa8246224c6c2fa03521"),
    "venueName" : "Oleron Villa",
    "date" : "August 11, 2017 at 02:00PM",
    "url" : "http://4sq.com/KBPBDW",
    "mapRef" : "http://maps.google.com/maps/api/\
      staticmap?center=45.87076061909858,\
      -1-1-1-1.2466156482696533 zoom=16&\
      size=710x440↦type=roadmap&sensor=false \
      markers=color:red%7C45.87076061909858,\
      -1.2466156482696533&\
      key=AIzaSyC2e-2nWNBM0VZMERf2I6m_PLZE4R2qAoM"
}
```

Constructing an iOS/Apple Watch App to automate workflows

The first example showed how to record a check-in into our own service as a side effect of checking into an existing service (Foursquare).

What if we wanted to create new, independent check-ins, from a mobile device? What if we also wanted to augment the check-ins with additional data? Another requirement could be to let our team know of the check-in through a Slack channel.

A valid approach would be to build a new mobile client using React Native or NativeScript. Slack and Google Maps have their own REST APIs and so the new App could certainly integrate with them in addition to our Mongopop API. Before investing in that development work, it would be great to prototype the concept and see if it proves useful.

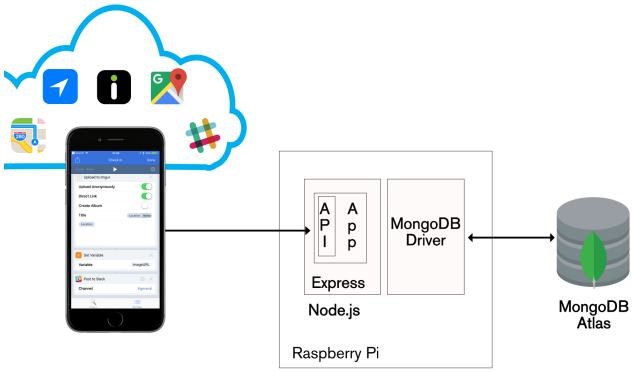
This is where we turn to the iOS **Workflow** app. Workflow has a number of similarities to IFTTT but there are also some significant differences:

- Workflow runs on your iOS device rather than in the cloud.
- Workflows are triggered by events on your iOS device (e.g. pressing a button) rather than an event in some cloud service.
- Workflow allows much more complex patterns than "**IF** **T**his event occurs in service A **T**hen trigger **T**hat action in service B"; it can loop, invoke multiple services, perform calculations, access local resources (e.g. camera and location information) on your device, and much more.

Both applications/Workflows that we build here can be run on an iPad, iPhone, or Apple Watch.

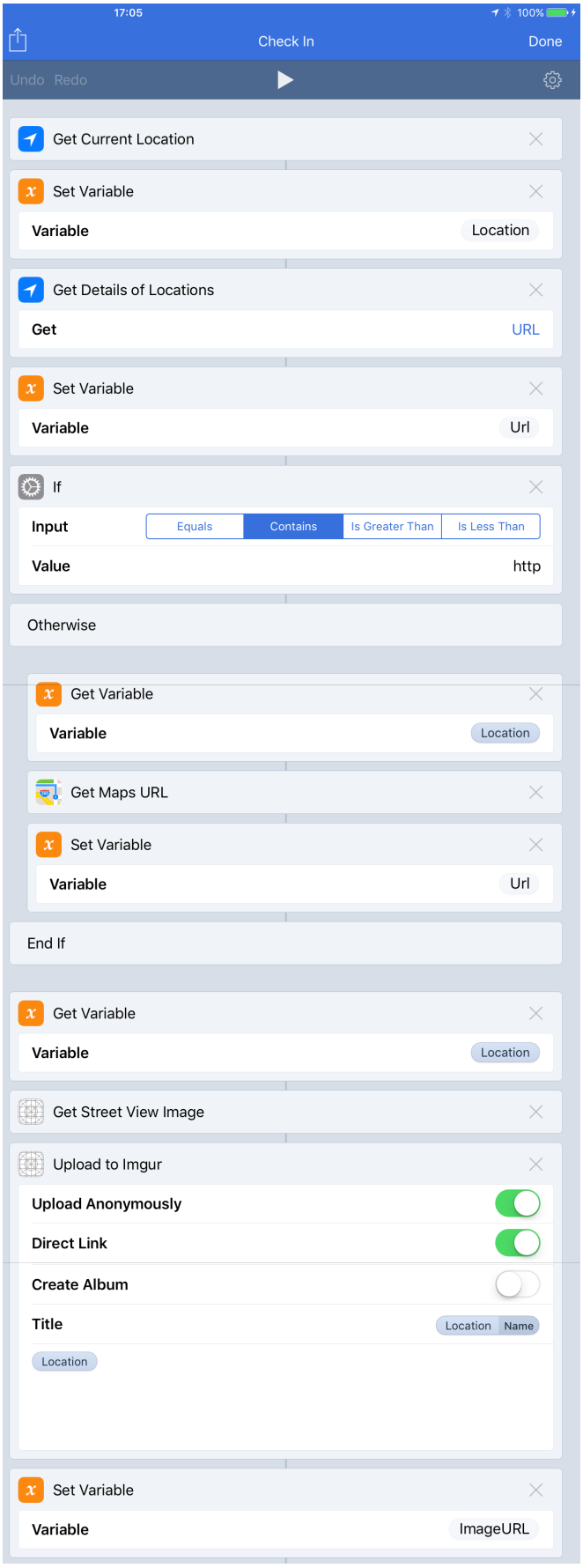
The first Workflow, *CheckIn*, performs these steps:

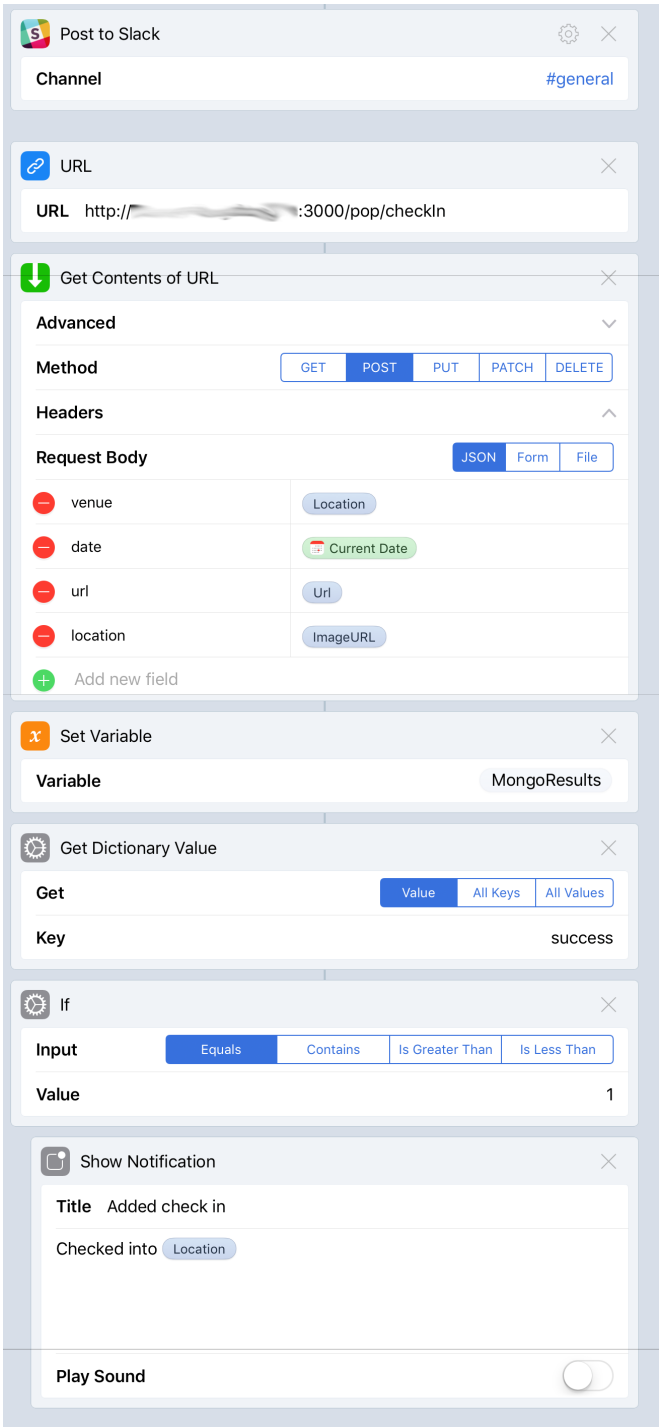
- Fetch the device's current location
- Retrieve any URL associated with that location
  - If none exists, fetch a map URL from Apple Maps
- Fetch a Google Street View image for the location
  - Upload this image to Imgur
  - Send the image URL to Slack
- Send a POST request to the `/pop/checkIn` Mongopop route
  - The request includes the location, date/time, URL (either from the venue or Apple Maps), and the StreetView image
- Post the location and URL to Slack
- Display error messages if anything fails

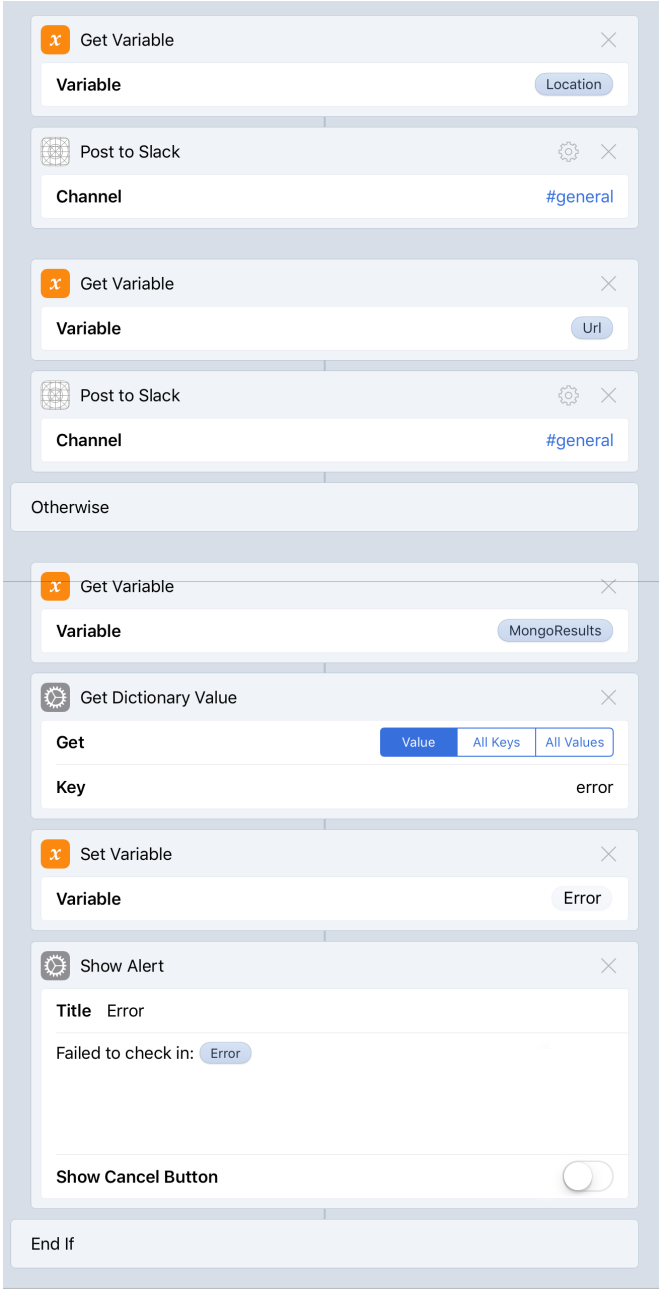


Implementing a Workflow involves dragging actions into the work area and then adding attributes to those actions (such as the address of the Mongopop API). The result of one action is automatically used as the input to the next action in the workflow. Results can also be stored in variables for use by later actions.

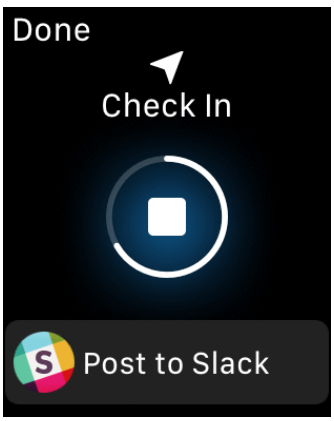
This is the *Check In* workflow:







The same app/workflow can be run from an Apple Watch:



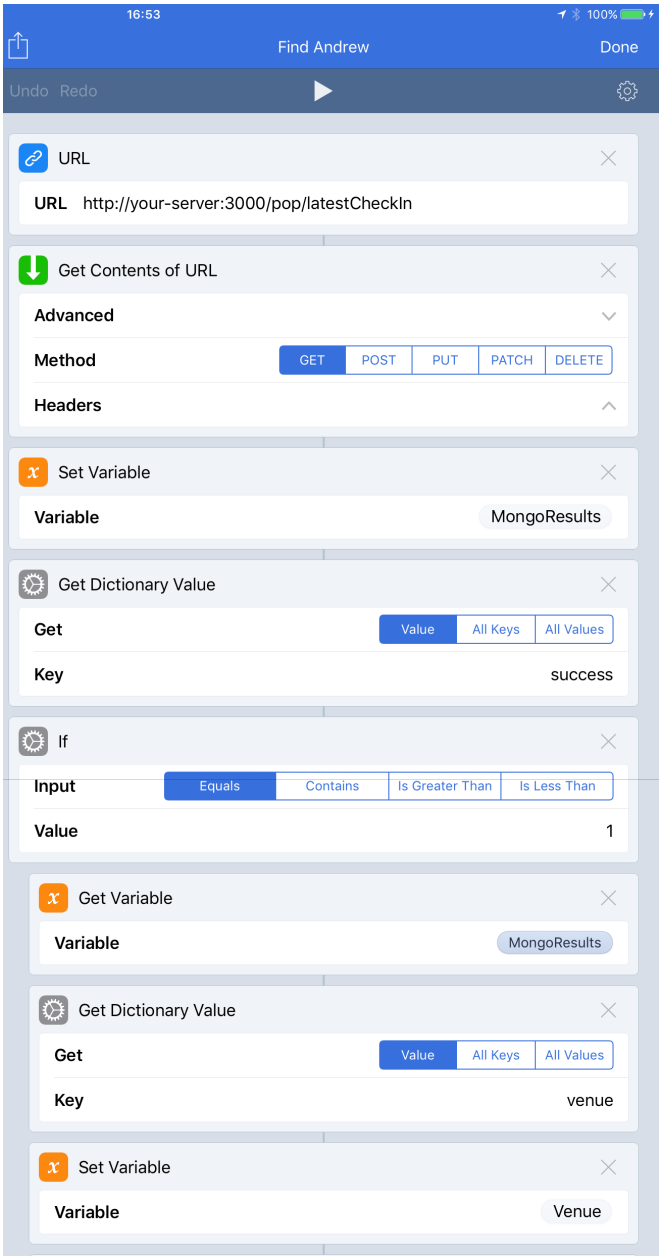
We can confirm that check-in record has been stored as a document in MongoDB Atlas (note that the database and collection names are defined in `config.js`):

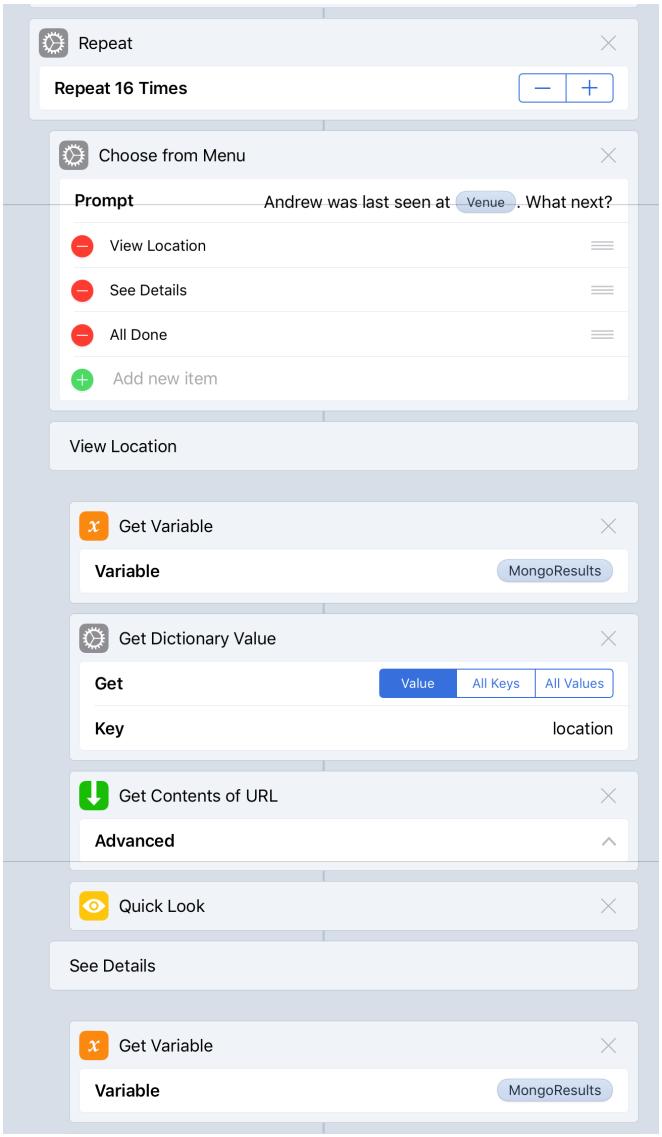
```
Cluster0-shard-0:PRIMARY> use maker
switched to db maker
Cluster0-shard-0:PRIMARY> db.foursq.find()
  .sort({_id: -1}).limit(1).pretty()
{
    "_id" : ObjectId("58c1505742067a03283be541"),
    "venueName" :
      "77-79 King St, Maidenhead SL6 1DU, UK",
    "date" : "9 Mar 2017, 12:53",
    "url" : "http://maps.apple.com/?q=77-79%20King%20St,\
      %20Maidenhead%20SL6%201DU,%20UK&\
      ll=51.520409,-0.722196",
    "mapRef" : "http://i.imgur.com/w3KyIVU.jpg"
}
```

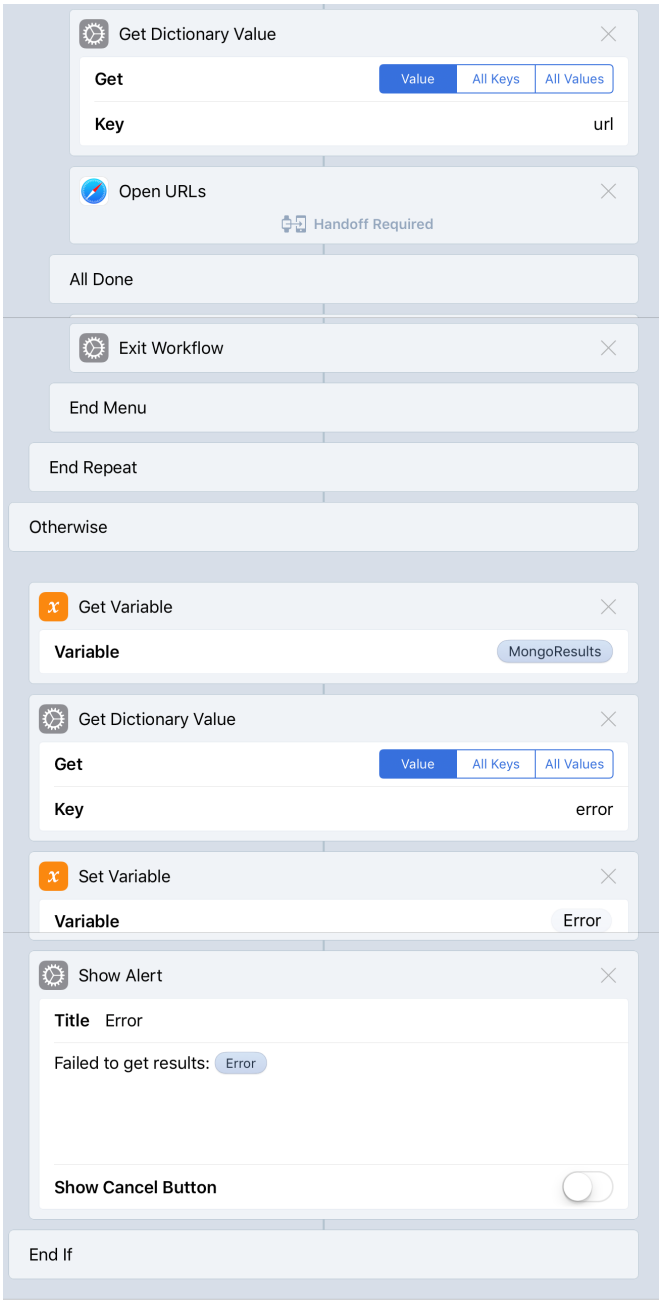
The second app/workflow retrieves and displays details of the most recent check-in. It performs these steps:

- Read from the `/pop/latestCheckIn` Mongopop REST API Route using GET.
- If the results indicate a successful operation then:
  - Extract the location from the results
  - Display the location and prompt the user if they'd like to:
    - See the location data (image)
    - Follow the location's URL (typically an Apple Maps link)
    - Finish
- If the Mongopop operation fails, display an appropriate error.

The full workflow is shown here:







View a video of running the app on an iPad.

The same app can be run from an Apple Watch:

Hands-free – Amazon Alexa Skills

Two of today's biggest industry trends are **machine learning** and **serverless computing**. Amazon's **Alexa** service (typically accessed through Amazon's **Echo** device) is at the forefront of both. In addition to interpreting voice commands for Amazon's own services (e.g. ordering more coffee beans or playing a particular song), developers can implement their own *skills*. Many are publicly available from 3rd parties such as Nest, Harmony, and Spotify, but you're free to implement and test your own – submitting it for review and public use when ready.

The business logic behind Alexa skills are typically run within Amazon's serverless computing service – **AWS Lambda**. Lambda is a great product for services that handle low or bursty levels of traffic – rather than paying a flat rate for a physical or virtual server, you pay a small fee for every event handled (and you even get a low-medium level of throughput for free). If your service really takes off then Lambda automatically scales out.

Assuming that you decide to use Lambda, there are three main components to your skill:

- The set of **intents** – instructions that a user can give to Alexa
- For each intent, a set of *utterances* that the user might say in order to signal that intent
- The actual logic which is invoked whenever the user signals an intent – implemented as a Lambda function

The **Mongo** Alexa skill has 3 intents/commands:

- **WhereIntent**: Find the most recent location that I checked in to
- **CountIntent**: Count how many times I've checked in
- **HelpIntent**: Explain what the available commands/ intents are

The intents are defined as a JSON document:

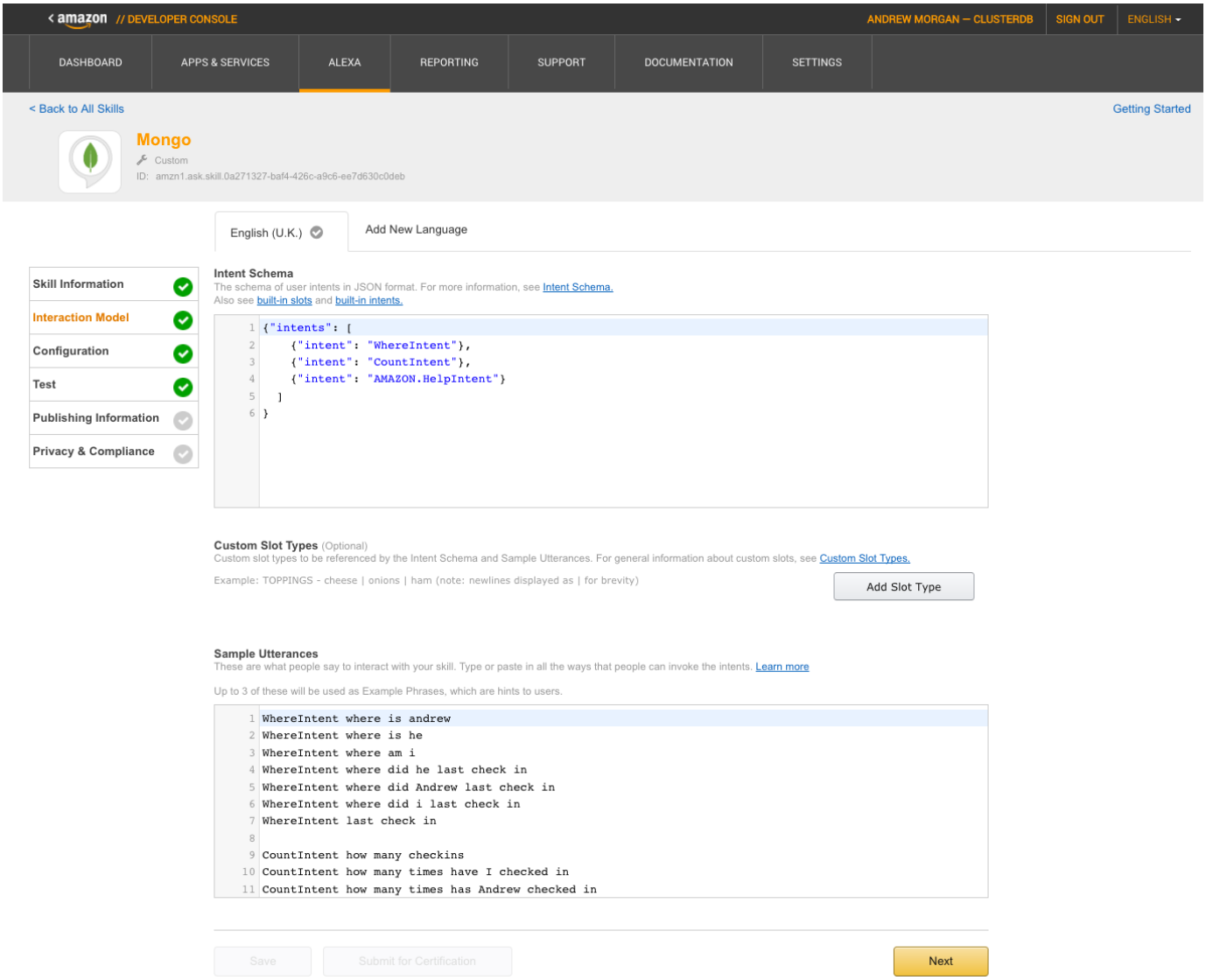
```
{"intents": [
  {"intent": "WhereIntent"},
  {"intent": "CountIntent"},
  {"intent": "AMAZON.HelpIntent"},
  ]
}
```

The utterances for each of those intents must also be defined:

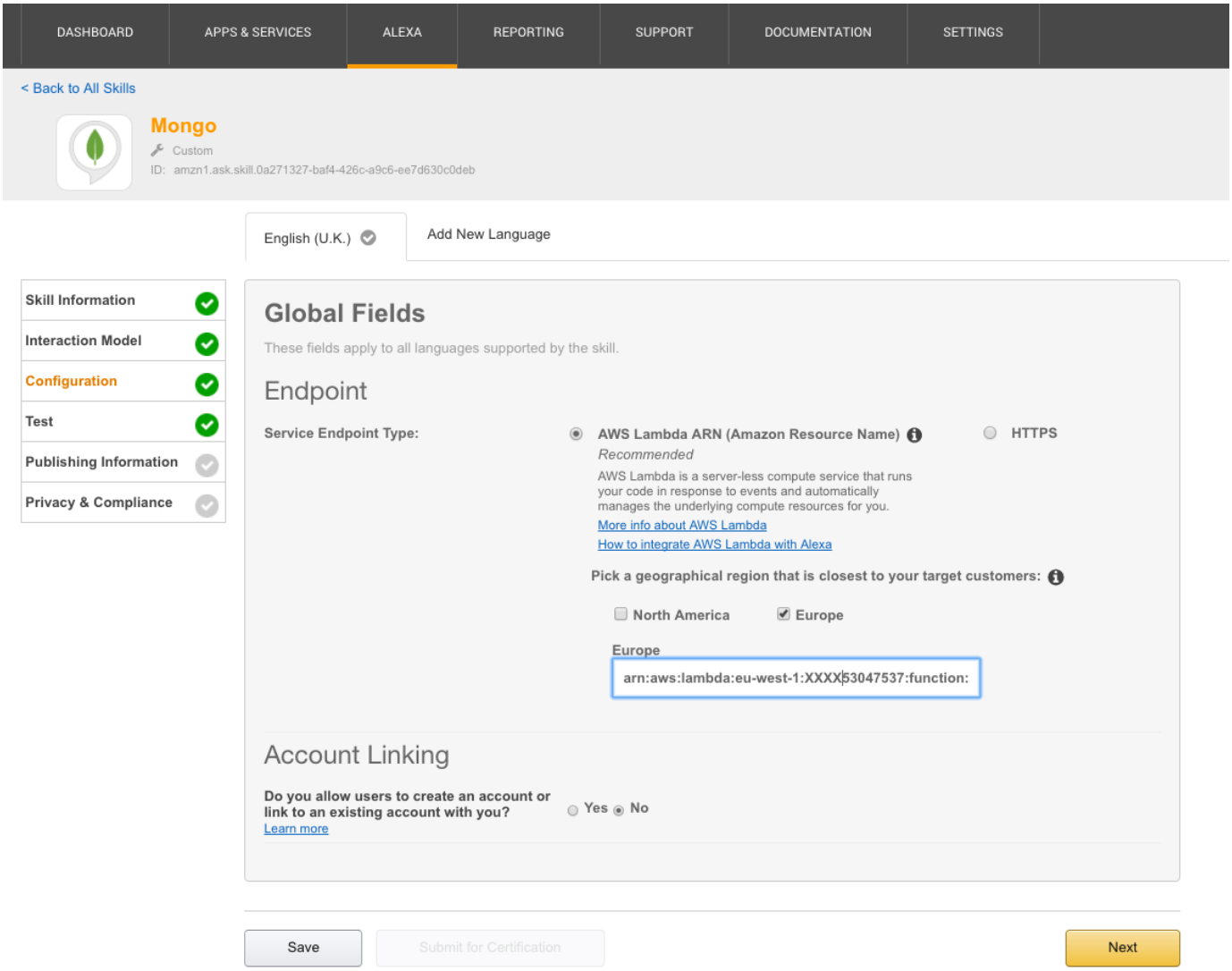
```
WhereIntent where is andrew
WhereIntent where is he
WhereIntent where am i
WhereIntent where did he last check in
WhereIntent where did Andrew last check in
WhereIntent where did i last check in
WhereIntent last check in

CountIntent how many checkins
CountIntent how many times have I checked in
CountIntent how many times has Andrew checked in
CountIntent how many times has he checked in
CountIntent how many check ins
CountIntent check in count
```

Note that no utterances need to be added for the `AMAZON.HelpIntent` as that intent is built in.

The skill is created in the Amazon Developer Console using the Alexa wizard; where the intentions and utterances can be added:



In the next screen, you indicate where the the skill's business logic runs; in this case, I provide the Amazon Resource Name (ARN) of my Lambda function:



The logic for the Mongo skill is implemented in the `index.js` file (part of the MongoDB-Alexa GitHub repository):

```
"use strict";

var Alexa = require("alexa-sdk");
var request = require("request");
var config = require("./config.js");

exports.handler = function(event, context, callback) {

    console.log("In handler");

    var alexa = Alexa.handler(event, context);
    alexa.appId = config.appId;

    alexa.registerHandlers(handlers);
    alexa.execute();
};

var handlers = {
  "AMAZON.HelpIntent": function () {
  this.emit(':tellWithCard',
    "Ask where Andrew is or ask how many times he\
     has checked in",
    "Mongo - Where's Andrew",
    "Ask where Andrew is or ask how many times he\
     has checked in",
    {
      smallImageUrl: "https://media.licdn.com/mpr/mpr/\
        shrinknp_200_200/AAEAAQAAAAAAAALQAAAAJDhkZWUxZDQ\
        xLTM0NjctNDcxZS04NmJiLTA1YzRhNGVlNWY0ZQ.jpg",
      largeImageUrl: "https://media.licdn.com/mpr/\
        mpr/shrinknp_200_200/AAEAAQAAAAAAAALQAAAAJDhkZW\
        UxZDQxLTM0NjctNDcxZS04NmJiLTA1YzRhNGVlNWY0ZQ.jpg"
    }
  )},
  "CountIntent": function () {

    console.log("In CountIntent");

    var countURL = config.mongopopAPI + "checkInCount";
    var _this = this;

    request({url: countURL, json: true},
        function (error, response, body) {
      console.log("in callback");
      if (error || response.statusCode != 200) {
        console.log("Failed to count checkins: " +
          error.message);
        _this.emit(':tellWithCard',
          "Network error, check Alexa app for\
            details",
          "Mongo - Error",
          "Network error: " + error.message,
          {
            smallImageUrl: "https://cdn3.iconfinder.com/\
              data/icons/wifi-2/460/\
              connection-error-512.png",
            largeImageUrl: "https://cdn3.iconfinder.com/\
              data/icons/wifi-2/460/\
              connection-error-512.png"
          }
        )
      } else {
        if (body.success) {

         var successString =
          "Andrew has checked in " + body.count +
          " times.";
          _this.emit(':tellWithCard',
            successString,
            "Mongo - Where's Andrew",
            successString,
            {
              smallImageUrl: "https://media.licdn.com/\
              mpr/mpr/shrinknp_200_200/AAEAAQAAAAAAAALQ\
              AAAAJDhkZWUxZDQxLTM0NjctNDcxZS04NmJiLTA1Y\
              zRhNGVlNWY0ZQ.jpg",
              largeImageUrl: "https://media.licdn.com/\
              mpr/mpr/shrinknp_200_200/AAEAAQAAAAA\
              AAALQAAAAJDhkZWUxZDQxLTM0NjctNDcxZS04\
              NmJiLTA1YzRhNGVlNWY0ZQ.jpg"
            }
          )
        } else {
          console.log("Failed to count checkins: "
            + body.error);
          _this.emit(':tellWithCard',
            "Application error, check Alexa app \
              for details",
            "Mongo - Error",
            "Application error: " + body.error,
            {
              smallImageUrl: "https://upload.wikimedia\
                .org/wikipedia/commons/thumb/b/bf/\
                 Sad_face.gif/1024px-Sad_face.gif",
              largeImageUrl: "https://upload.wikimedia\
                .org/wikipedia/commons/thumb/b/bf/\
                Sad_face.gif/1024px-Sad_face.gif"
            }
          )
        }
      }
    })
  },
  "WhereIntent": function () {

    console.log("In WhereIntent");

    var latestCheckinURL = config.mongopopAPI +
      "latestCheckin";
    var _this = this;

    request({url: latestCheckinURL, json: true},
        function (error, response, body) {
      console.log("in callback");
      if (error || response.statusCode != 200) {
        console.log("Failed to fetch latest\
         Checkin, network problem: " +
         error.message);
        _this.emit(':tellWithCard',
          "Network error, check Alexa app for\
           details",
          "Mongo - Error",
          "Network error: " + error.message,
          {
              smallImageUrl: "https://cdn3.iconfinder\
              .com/data/icons/wifi-2/460/connection-\
              error-512.png",
              largeImageUrl: "https://cdn3.iconfinder\
              .com/data/icons/wifi-2/460/\
              connection-error-512.png"
          }
        )
      } else {
        if (body.success) {
          var successString =
```

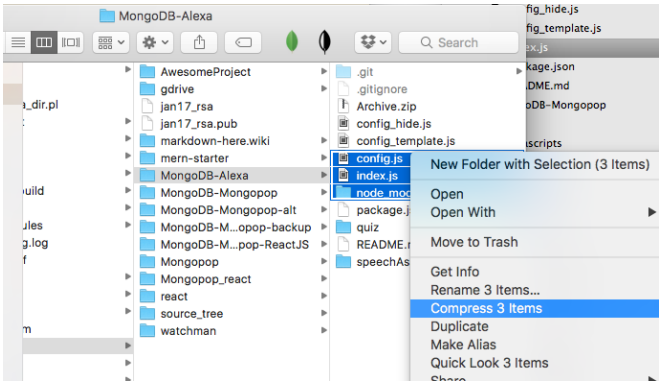
```
            "Andrew last checked in to " +
              body.venue + " on " + body.date;
            var imgURL = body.location.replace(
              "http", "https");
            _this.emit(':tellWithCard',
              successString,
              "Mongo - Where's Andrew",
              successString + ". Location URL: " +
                body.url + ". View venue: " +
                body.location + ".",
              {
                smallImageUrl: imgURL,
                largeImageUrl: imgURL
              }
            )
          } else {
            console.log("Failed to fetch latest\
             checkin, app error: " + body.error);
            _this.emit(':tellWithCard',
              "Application error, check Alexa app\
               for details",
              "Mongo - Error",
              "Application error: " + body.error,
              {
                smallImageUrl: "https://upload.wikimedia.\
                org/wikipedia/commons/thumb/b/bf/Sad_face.\
                gif/1024px-Sad_face.gif",
                largeImageUrl: "https://upload.wikimedia.\
                org/wikipedia/commons/thumb/b/bf/Sad_face\
                .gif/1024px-Sad_face.gif"
              }
            )
          }
        }
      })
    },
    "Unhandled": function () {
      this.emit(':tellWithCard',
        "Unhandled event",
        "Mongo - Error",
        "Unhandled event",
        {
          smallImageUrl: "https://upload.wikimedia.org/\
            wikipedia/commons/thumb/b/bf/Sad_face.gif/\
            1024px-Sad_face.gif",
          largeImageUrl: "https://upload.wikimedia.org/\
            wikipedia/commons/thumb/b/bf/Sad_face.gif/\
            1024px-Sad_face.gif"
        }
      )
    }
};
```

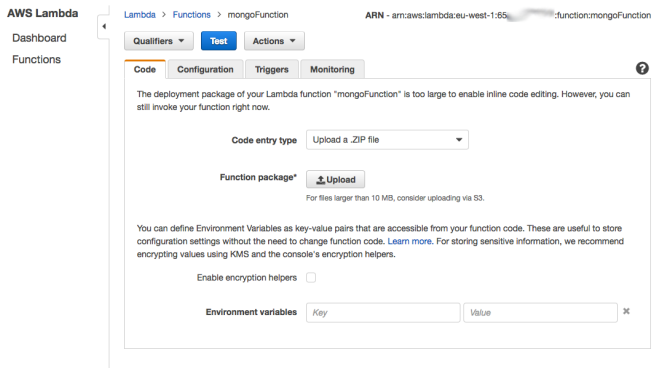
As explained earlier, the aim of this chapter is to cover a broad set of technologies rather than going too deeply into any one but explaining a few concepts may help you understand what this code is doing:

- A **handler** is implemented for each of the intents; that handler is invoked when the user speaks one of the utterances associated with that intent
- The handlers for the `CountIntent` and `WhereIntent` makes calls to the Mongopop REST API using the `request` function
- The **emit** method is how the handlers can send results or errors back to the user (via Alexa)
- The **card**, referred to by `tellWithCard`, is visual content (text and images) that are displayed in the Alexa app

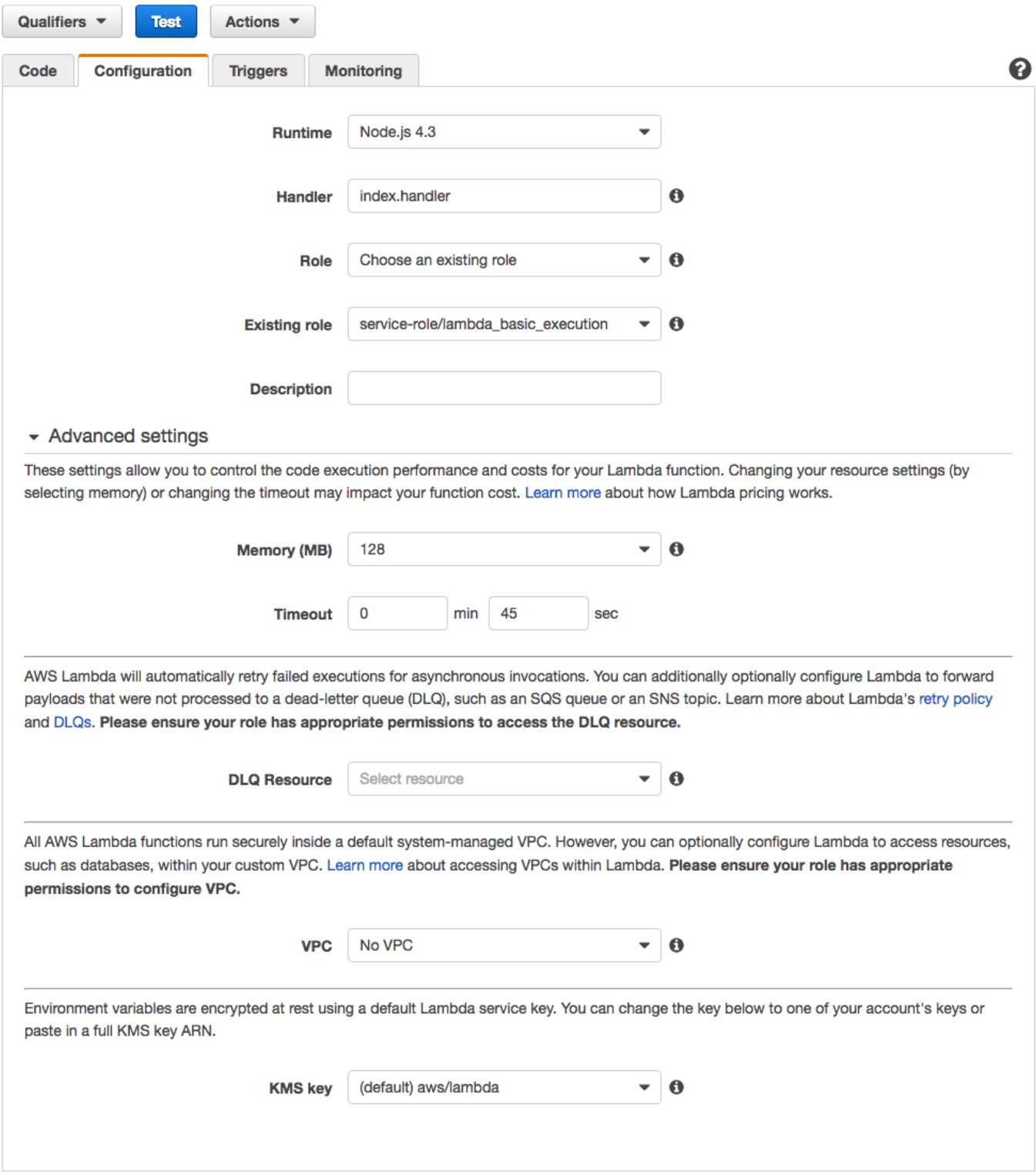
Note that this is a simple skill which receives a request and sends a single response. It is also possible to implement an interactive state machine where there's a conversation between the user and Alexa - in those skills, the logic uses both the latest intent and the past context in deciding how to respond. Note that the Lambda function is always stateless and so all data should be stored in a database such as MongoDB.

The skill is deployed to AWS Lambda through the AWS Management Console. The `index.js`, config.js and `node_modules` directory (created by running `npm install`) should be archived into a single Zip file which is then uploaded to AWS:

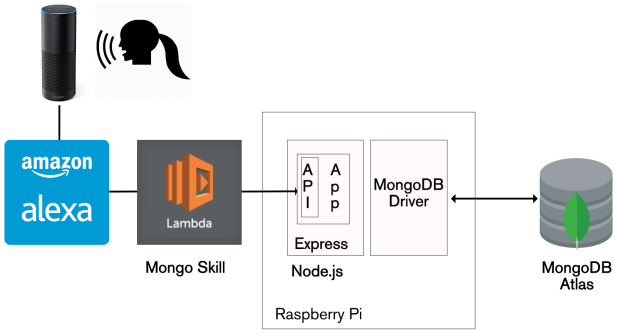




There are a number of extra configuration options – such as the runtime environment to use (Node.js), the user role, the amount of memory to be made available to the function, and how long each invocation of the function should be allowed to run (the function is making external HTTP requests and so it may need a few seconds):



As a reminder, the user speaks to the Amazon Echo device, then the Alexa application invokes an AWS Lambda function, which implements the business logic for the Mongo skill ,which then interacts with the MongoDB database via the Mongopop REST API:



To start, test the simplest intent – asking the Mongo skill for help with "Alexa, ask Mongo for help"; see the results.

Note that the visual card can contain more information than Alexa's spoken response. For example, if there is an error in the Mongopop backend, the returned error message is displayed on the card.

Next, we can ask Alexa how many times I've checked in and where my last check-in was. Note that I could have used any of the utterances associated with these intents (and Alexa will automatically convert similar phrases). See the results in this video.

Chapter Summary

This chapter explored some alternative ways to build client applications; in particular, it showed how to combine existing cloud services with a bit of new logic to create something brand new. We looked at a number of technologies to help build applications quickly and efficiently:

- IFTTT: Make events in one cloud service trigger actions in another
- Workflow: Automate complex tasks involving multiple services on an iOS device
- Amazon Alexa: Implement your own voice-controlled skills
- AWS Lambda: Host and scale your business logic in the cloud while only paying for the transactions you process

Increasingly, applications leverage multiple services (if only to allow the user to share their efforts on different social media networks). The key to all of these integrations is the REST APIs provided by each service.

MongoDB Stitch – the latest, and best way to build your app

So far in this book, I've stepped through how you can build modern applications on a stack of MongoDB, Node.js, Express, and Angular or React. What if there was a service that took care of everything apart from from the application frontend (Angular, React, or other technology)? MongoDB Stitch is that service, it's a new backend as a service (BaaS) for MongoDB applications.

The purpose of this chapter is to introduce what MongoDB Stitch is and, most importantly, demonstrate exactly how you use it – both configuring your app through the Stitch backend UI, and invoking that app backend from your frontend code or other services.

What is MongoDB Stitch?

MongoDB Stitch is a BaaS, giving developers a REST-like API to MongoDB, and composability with other services, backed by a robust system for configuring fine-grained data access controls.

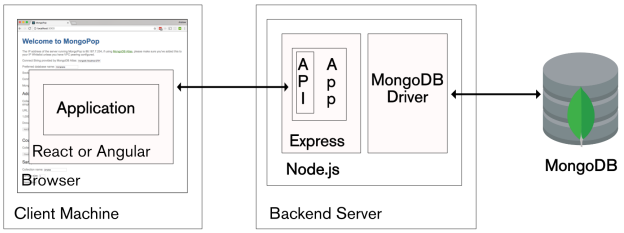
Stitch allows developers to focus on building applications rather than on managing data manipulation code or service integration. As application and display logic continues to move into the frontend web and mobile tiers, the remaining backend is often dominated by code to handle storing and retrieving data from a database, security, data privacy, and integrating various services. MongoDB Stitch provides that same functionality but declaratively rather than using boilerplate backend code.

The data access rules in MongoDB stitch are entirely declarative and designed to be expressive enough to handle any application, including sensitive data such as payment details. For a given collection, you can restrict

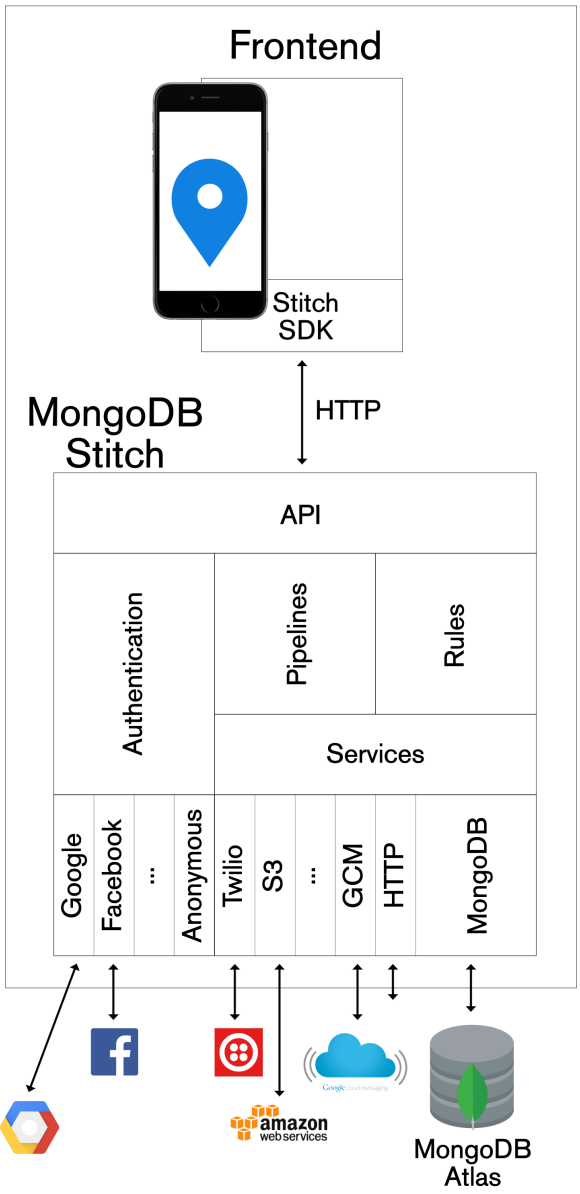
what operations are permitted and what fields can be accessed – according to user id, role, or custom criteria. Access can even be limited to specific aggregations – allowing analysts to work with the data without exposing any individual's private information.

If you already have data in MongoDB, you can start by safely exposing it to new applications via Stitch's API – perhaps allowing read access to specific fields. You can authenticate users through built-in integrations auth providers.

Earlier, I detailed how to work with the technologies that are typically used to make up a modern application backend: MongoDB for the database, Node.js to run the backend logic, and a framework such as Express to provide a REST API:



Stitch, greatly simplifies your development and ops efforts for new applications by providing the entire backend as managed service. Even your frontend application code is simplified, as Stitch provides idiomatic SDKs for JavaScript, iOS, and Android – so you don't need to code HTTP requests directly. Further, the SDK/API isn't limited to just accessing MongoDB data, you can also use it for any other service registered with your Stitch application backend.

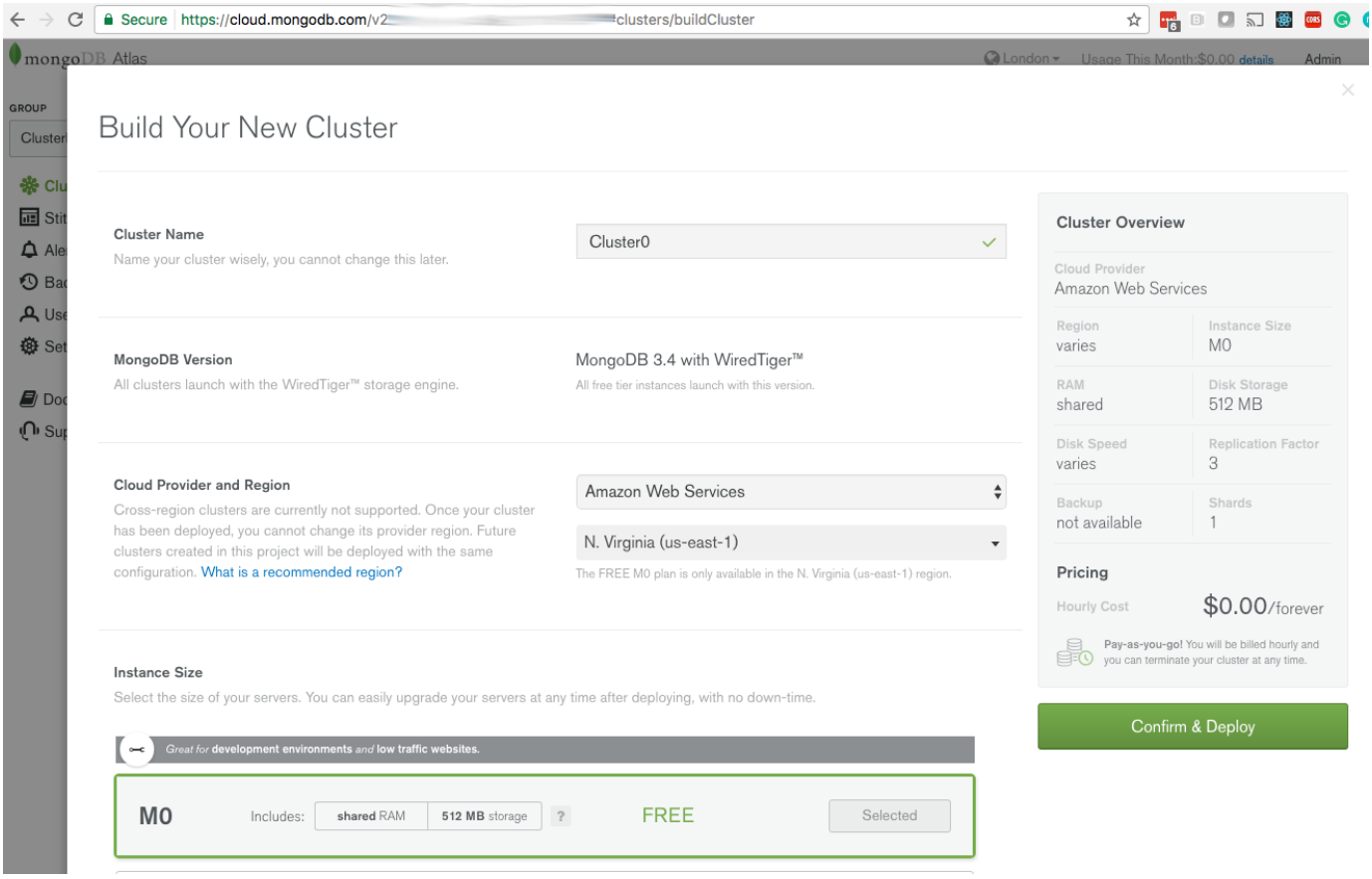


Building an application with MongoDB Stitch

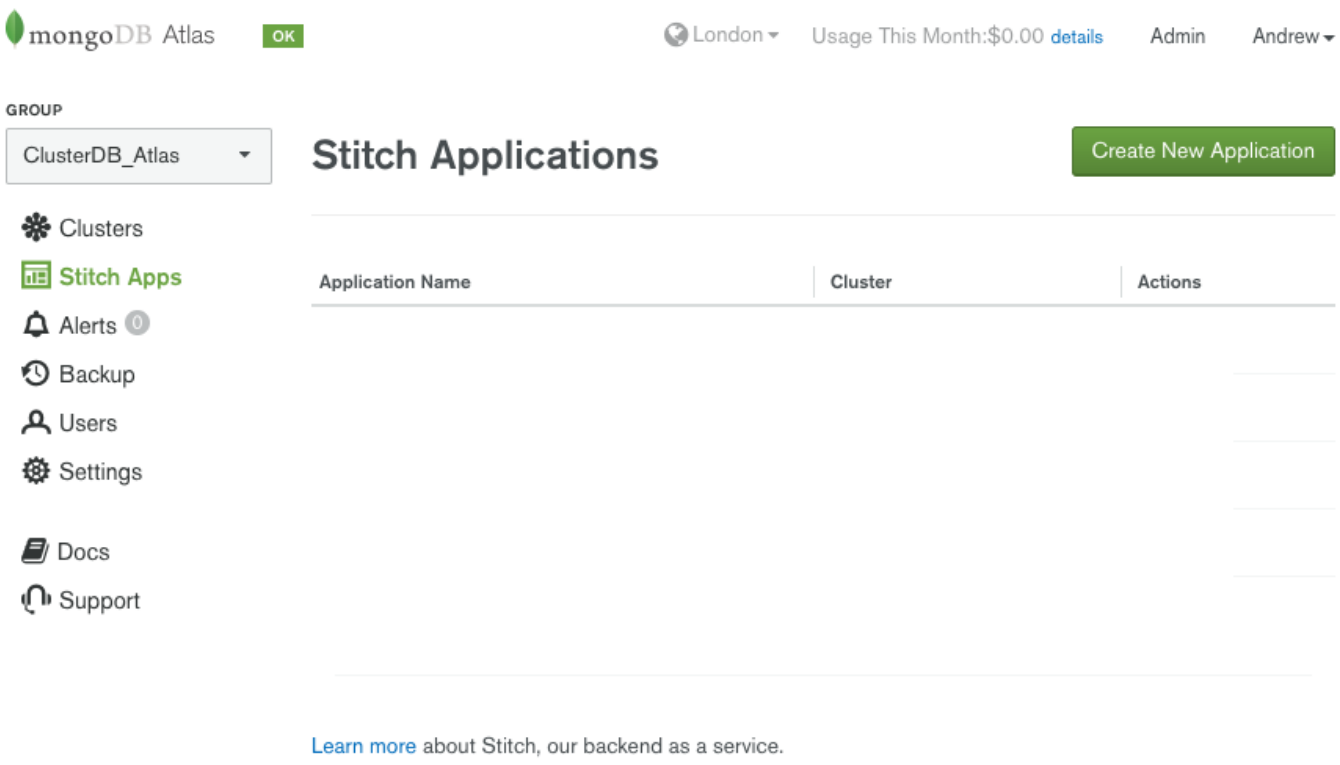
You can get started with MongoDB Stitch for free – use it with your free MongoDB Atlas cluster.

Creating your application in MongoDB Stitch

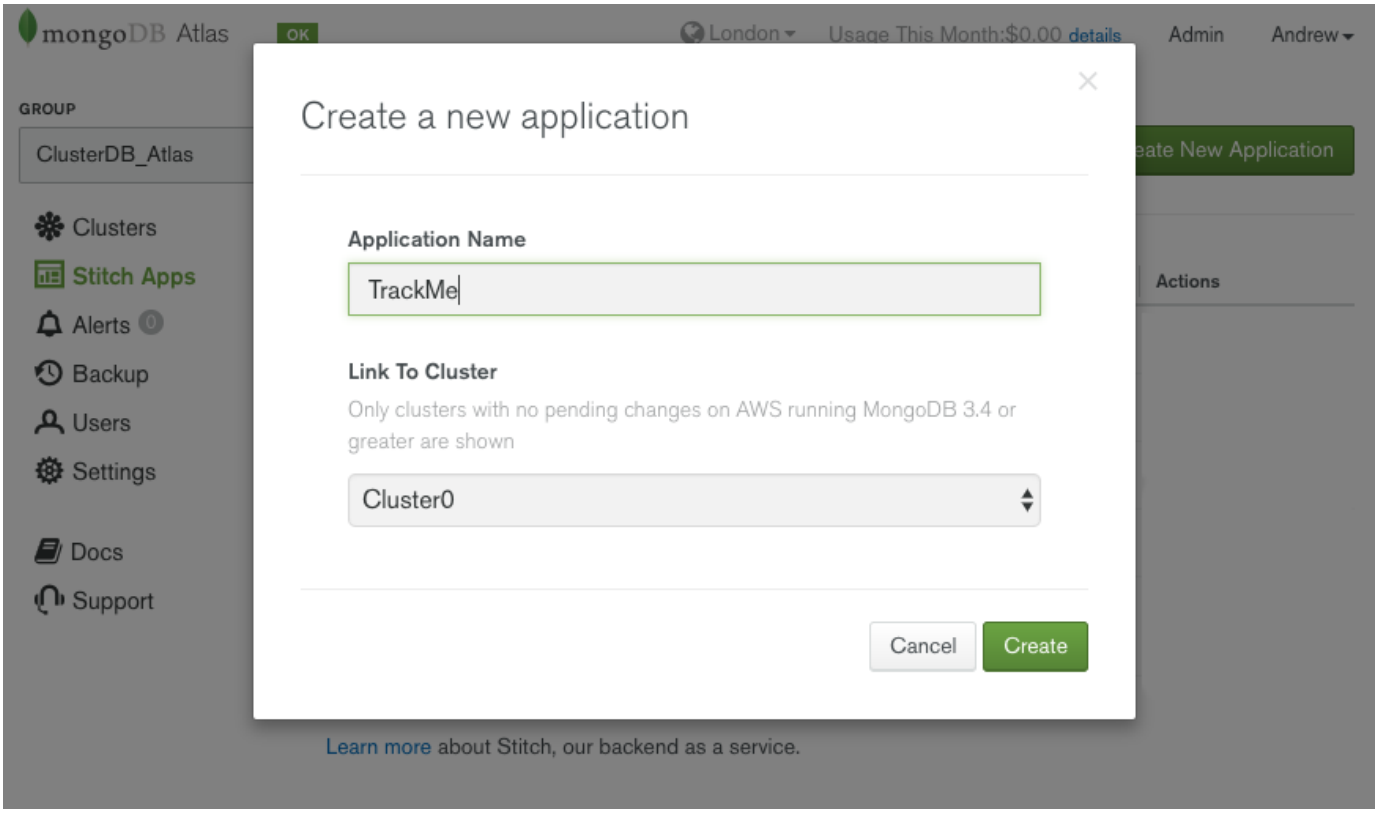
If you haven't done so already, create a new MongoDB Atlas cluster, selecting the *M0* instance type for the free tier:



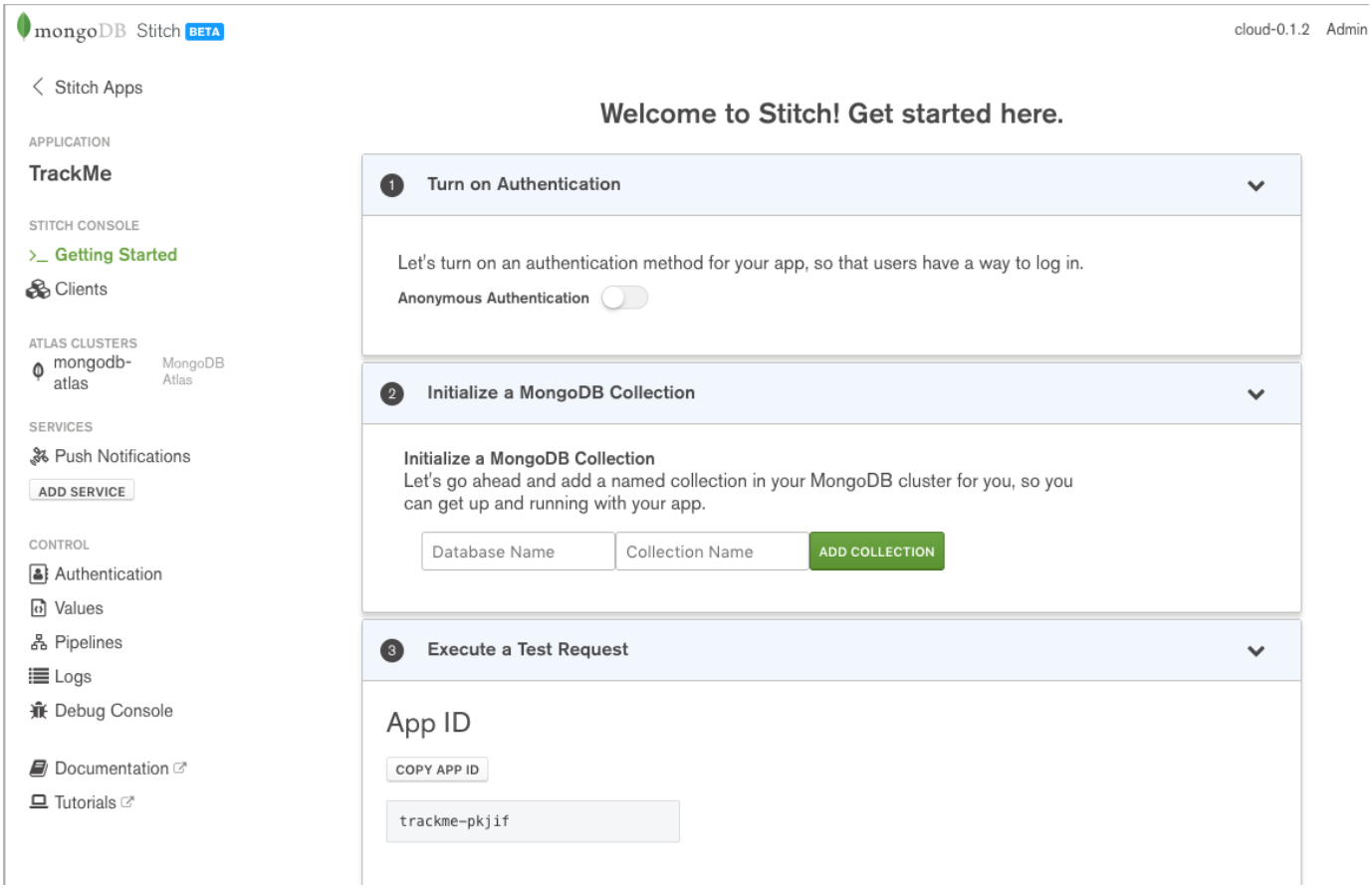
After the cluster has spun up, click on *Stitch Apps* and then *Create New Application*:



Give the application a name and ensure that the correct Atlas cluster is selected:

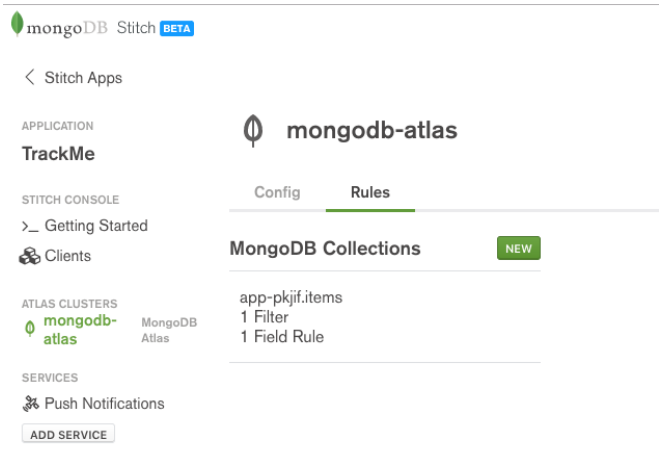


Once you've created the application, take a note of its *App ID* (in this example `trackme-pkjif`) as this will be needed by your application's frontend:

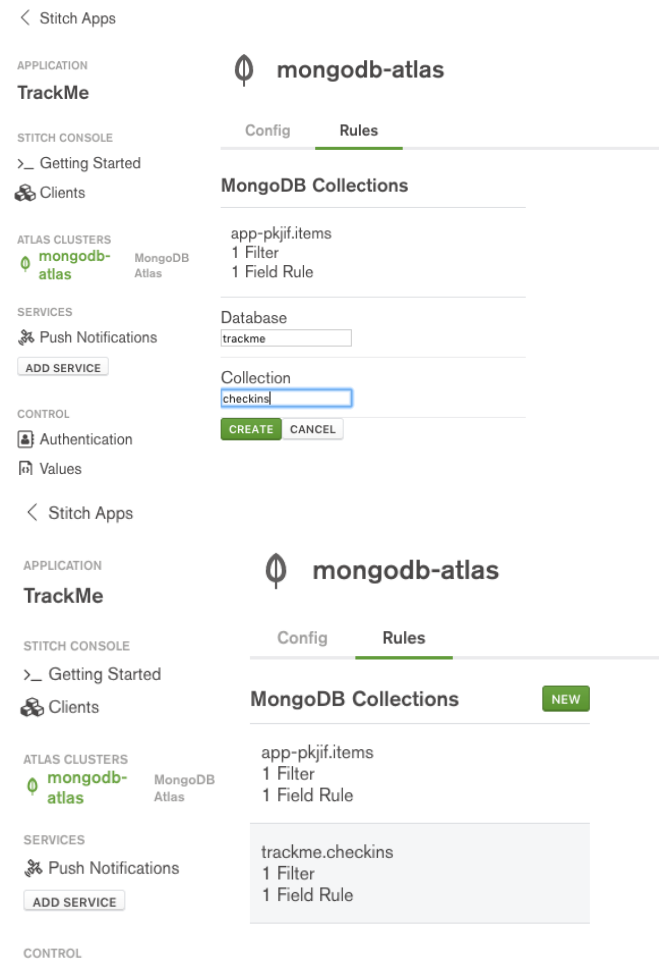


Backend database and rules

Select the *mongodb-atlas* service, followed by the *Rules* tab – this is where you define who can access what data from the MongoDB database:



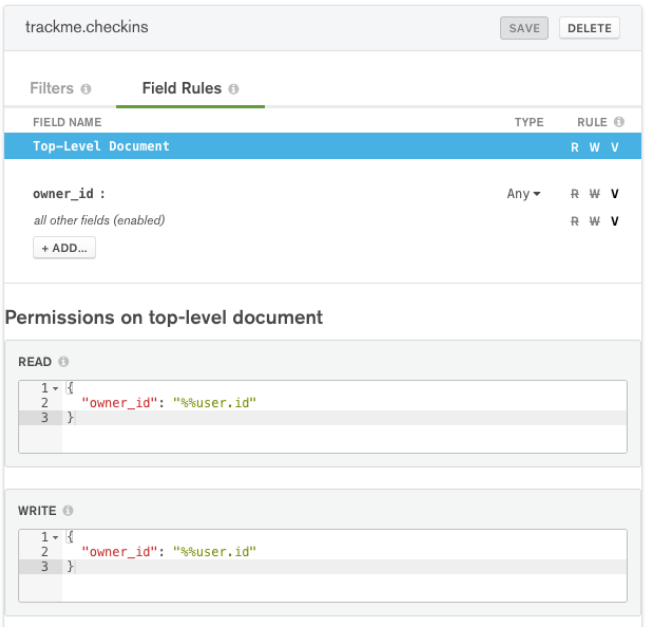
Set the database name to `trackme` and the collection to `checkins`:



A typical record from the `track.checkins` collection will look like this:

```
db.checkins.find().sort({_id: -1}).skip(2)
  .limit(1).pretty()
{
    "_id" : ObjectId("597f14fe4fdd1f5eb78e142f"),
    "owner_id" : "596ce3304fdd1f3e885999cb",
    "email" : "me@gmail.com",
    "venueName" : "Starbucks",
    "date" : "July 31, 2017 at 12:27PM",
    "url" : "http://4sq.com/LuzfAn",
    "locationImg" : "http://maps.google.com/maps/\
      api/staticmap?center=51.522058,-0.722497&\
      zoom=16&size=710x440↦type=roadmap&\
      sensor=false▮s=color:red%7C51.522058,\
      -0.722497&\
      key=AIzaSyC2e-2nWNBM0VZMERf2I6m_PLZE4R2qAoM"
}
```

Select the *Field Rules* tab and note the default read and write rules for the *Top-Level* document:



The default read rule is:

```
{
  "owner_id": "%%user.id"
}
```

The meaning is that a document can only be read from this collection if its `owner_id` field matches the `id` of the application user making the request (i.e. a user can only read their own data). `%%user` is an expansion which gives the rule access to information about the application end-user making the request – here we're interested in their unique identifier (`id`). Whenever a user adds a document to a collection, Stitch will set the `owner_id` to the ID of that user.

Overwrite the write rule with the following, then press *SAVE*:

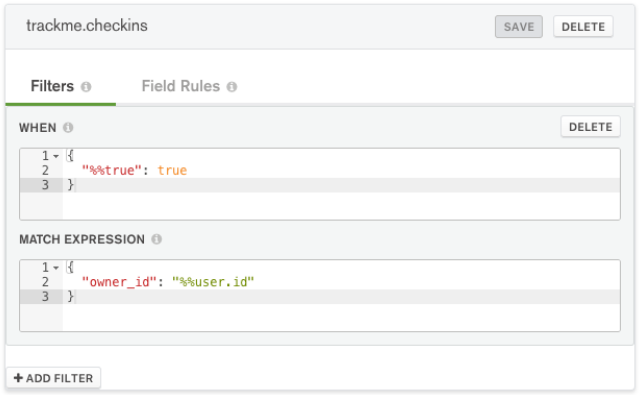
```
{
  "%or": [
    {
      "%%prevRoot.owner_id": "%%user.id"
    },
    {
      "%%prevRoot": {
        "%exists": 0
      }
    }
  ]
}
```

`%%prevRoot` is another expansion, representing the state of the document before the operation. You can read the above logic as: "Allow the write to succeed if either the the same user previously added the document or the document didn't exist (i.e. it's an insert)".

In addition to general rules for the document, read/write rules can be added for individual fields. Select the `owner_id` field and ensure that the validation rule is set to:

```
{
  "%or": [
    {
      "%%prev": "%%user.id"
    },
    {
      "%%prev": {
        "%exists": false
      }
    }
  ]
}
```

Filters control which documents a user sees when viewing a collection:



Ensure that *When* == `{"%%true": true}` and *Match Expression* == `{"owner_id": "%%user.id"}`. This means that the filter is always applied and that a user will only see their own documents.

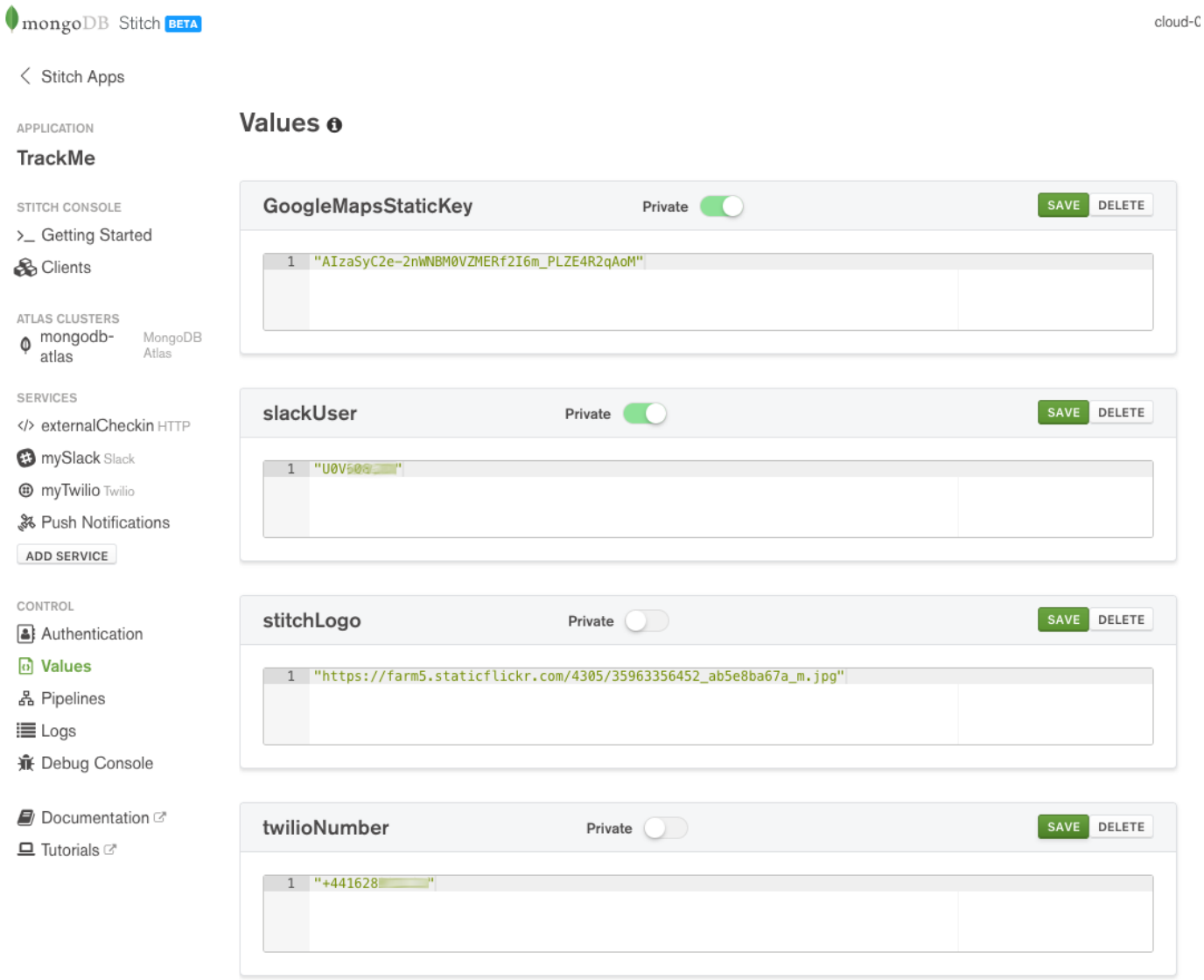
You should also add rules, the `trackme.users` collection, where a typical document will look like:

```
> db.users.findOne()
{
    "_id" : ObjectId("596e354f46224c3c723d968a"),
    "owner_id" : "596ce47c4fdd1f3e88599ac4",
    "userData" : {
        "email" : "andrew.morgan@mongodb.com",
        "name" : "Andrew Morgan",
        "picture" : "https://lh4.googleusercontent\
          .com/-lCBSTZFxhw0/AAAAAAAAAAI/AAAAAAAAAB4/\
          vX9Sg4dO8xE/photo.jpg"
    },
    "friends" : [
        "billy@gmail.com",
        "granny@hotmail.com"
    ]
}
```

Setup `trackme.users` with the same rules and filters as `trackme.checkins`.

Values/constants

Stitch provides a simple and secure way to store values associated with your application – a perfect example is your keys for public cloud services. Set up the following values:



By default, your WebHooks, named pipelines, and frontend application code can read the values. By setting the value to be private, you prevent access from your frontend code (or any other users of the Stitch API). The example React frontend code refers to the `twilioNumber` value (`%%values.twilioNumber`) when building a pipeline (if you wanted to keep the value more secure then you could implement a named pipeline to send the Twilio message and make `twilioNumber` private):

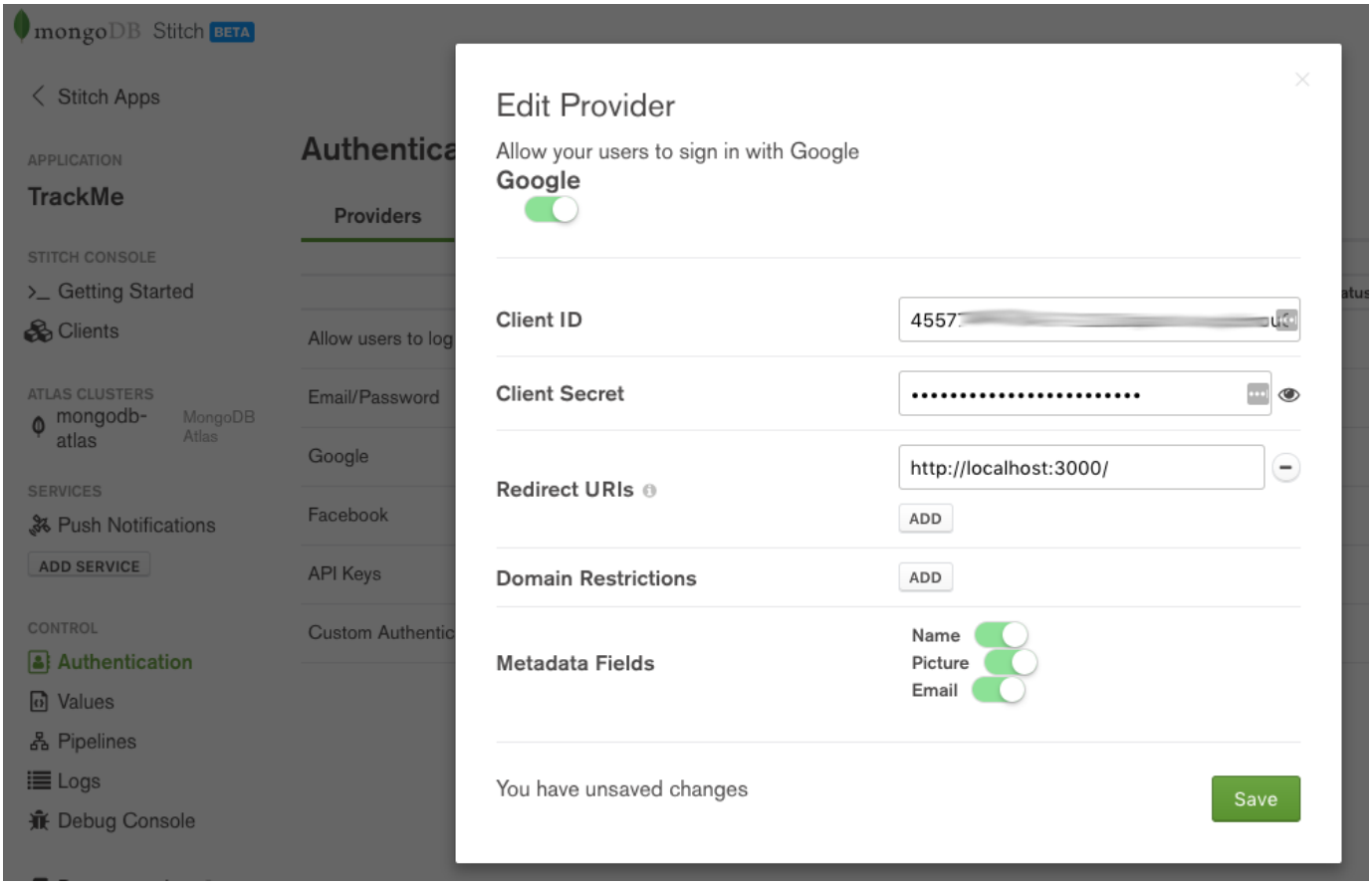
```
this.props.stitchClient.executePipeline([
  {
    service: "myTwilio",
    action: "send",
    args: {
      to: this.state.textNumber,
      from:
        "%%values.twilioNumber",
          // Relies on twilioNumber not being
          // private
      body: name + " last checked into "
        + venue
    }
  }
])
```

Authentication providers

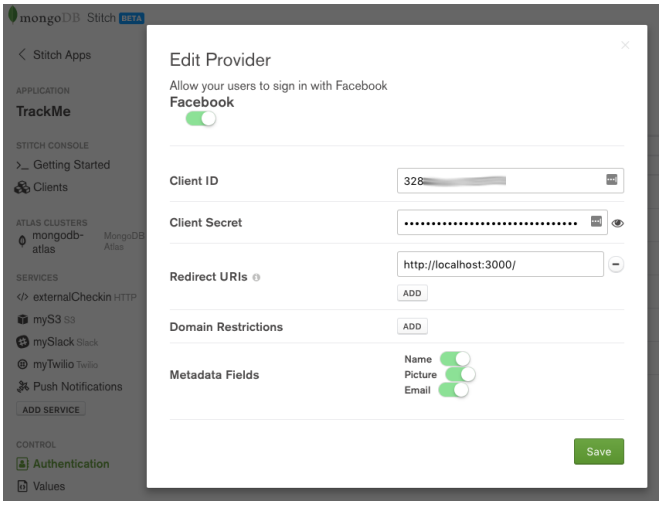
A key feature of Stitch is authenticating your app's end users – after which you can configure precisely what data and services they're entitled to access (e.g., to view documents that they created through their actions in the app). The following types of authentication are all supported:

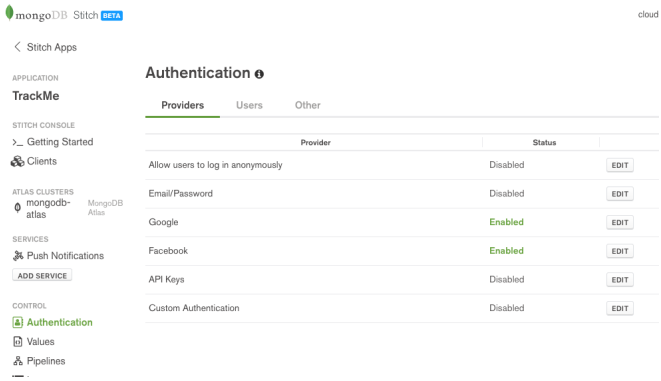
- **Anonymous** (the user doesn't need to register or log in, but they're still assigned an ID which is used to control what they see)
- **Email/Password**
- **Google**
- **Facebook**
- **Custom** (using JSON web tokens)

From the Authentication section of the Stitch UI, turn on Google authentication, providing the *Client ID* and *Client Secret* generated by Google. If you are running your app on your local machine then add `http://localhost:3000/` as a *Redirect URI*; if hosting externally, add the DNS hostname. Enable *Name*, *Picture*, and *Email* so that your app has access to those user credentials from Google. Click *Save*:



Turn on Facebook authentication, providing the *Client ID* and *Client Secret* generated by Facebook. If you are running your app on your local machine then add `http://localhost:3000/` as a *Redirect URI*; if hosting externally, add the DNS hostname. Enable *Name*, *Picture*, and *Email* so that your app has access to those user credentials from Facebook. Click *Save*.

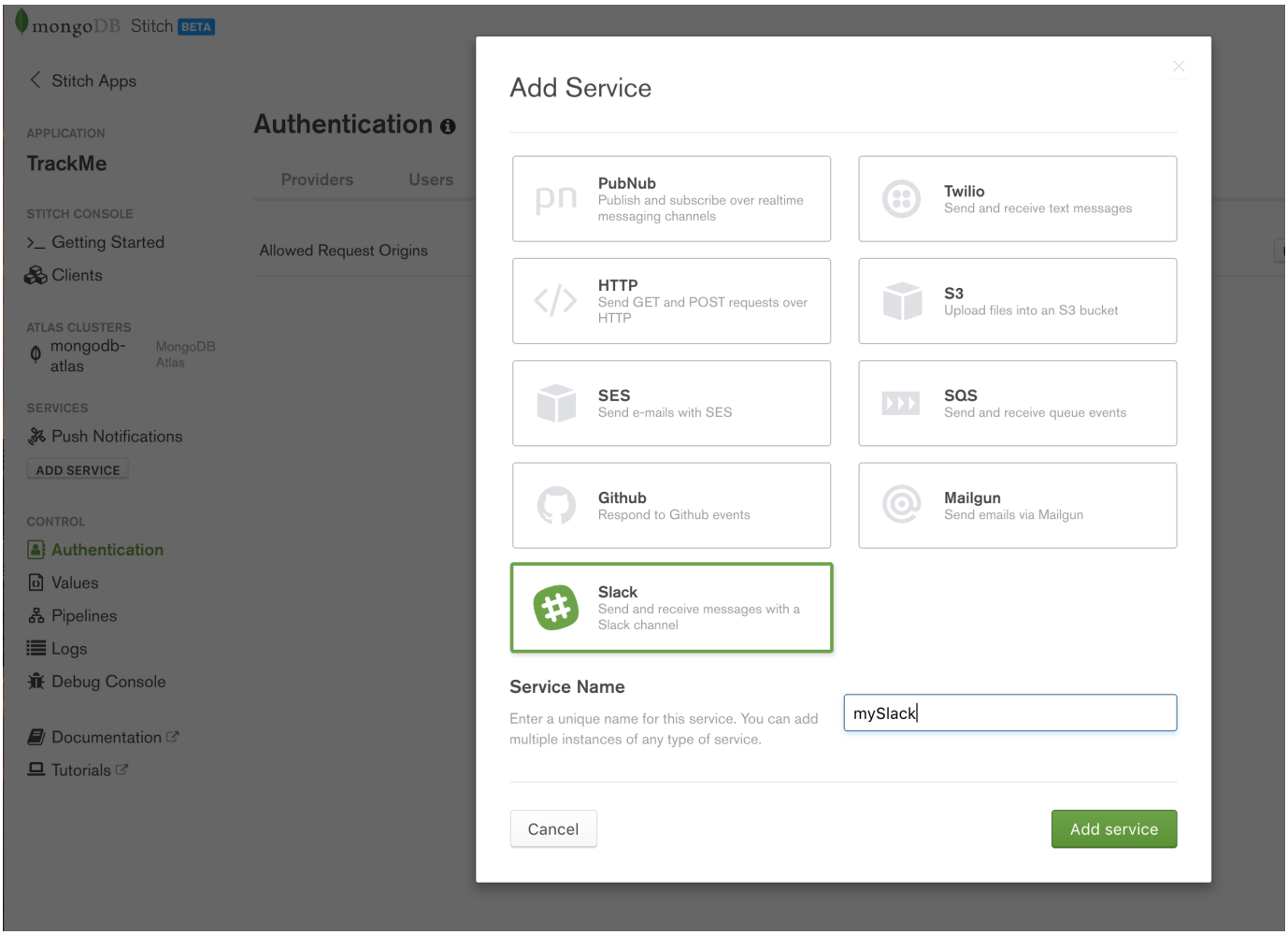




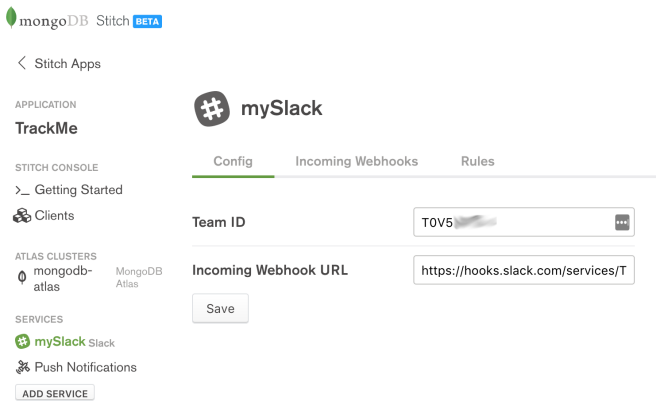
Adding other services (Slack & Twilio)

Stitch has some services pre-integrated, for others, you can use the HTTP Service.

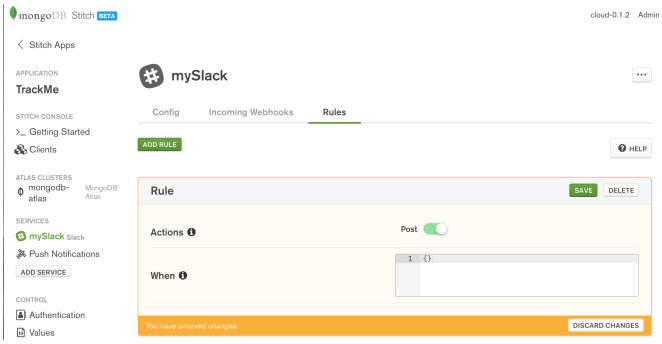
When a user checks in, a notification will be sent to a Slack channel using Stitch's Slack Service. Click on *Add Service* and then select *Slack*, name the service `mySlack` (your pipelines and WebHooks can refer to the service using that name), and then click *Add service*.



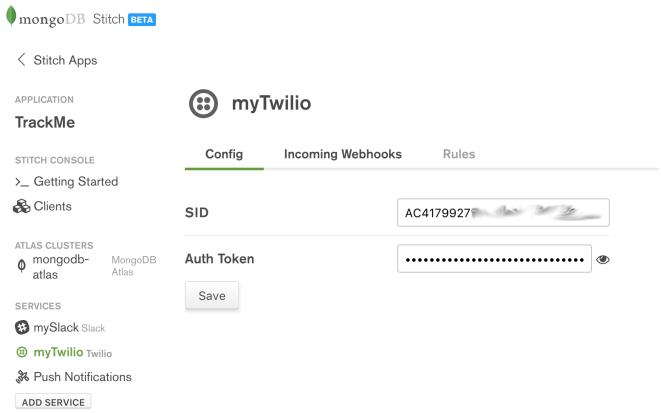
In the *Config* tab, enter the *Team ID* and *Incoming WebhookURL* provided by Slack:



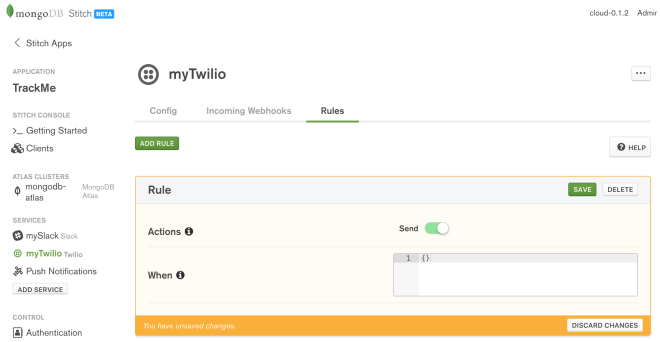
There is no need to add any WebHooks (the app will send out Slack messages but will not receive any). On the *Rules* tab, enable *Post* (as the Stitch app must use the HTTP POST method to send messages to Stitch's API), and then *Save*:



From the React web app, a logged-in user has the option to send an SMS text message, containing their latest check-in, to a phone number of their choice. To enable that service, you must configure the Twilio Service through the Slack UI:



The values to use for the *SSID* and the *Auth Token* can be retrieved after registering with Twilio. As with Slack, the app will not accept incoming messages from Twilio, and so there is no need to define any incoming WebHooks. In the *Rules* tab, enable the *Send* action and click *Save*:



Named service pipelines

Service pipelines are used to execute a sequence of actions. As with the Stitch features you've already seen, pipelines are defined using JSON documents. You can create pipelines on the fly in your application code, or you can preconfigure Named Pipelines. The advantages of named pipelines are:

- Enhanced security: access to secret resources, such as API keys, can be encapsulated within the Stitch backend. The alternative is to code them in the device-side code, where a user may attempt to reverse-engineer them.
- Code re-use: you can create the named pipeline once in the Stich backend, and then invoke it from multiple frontend locations (e.g., from multiple places in a web app, as well as from iOS and Android apps).
- Simpler code: keep the frontend application code clean by hiding the pipeline's details in the Stitch backend.

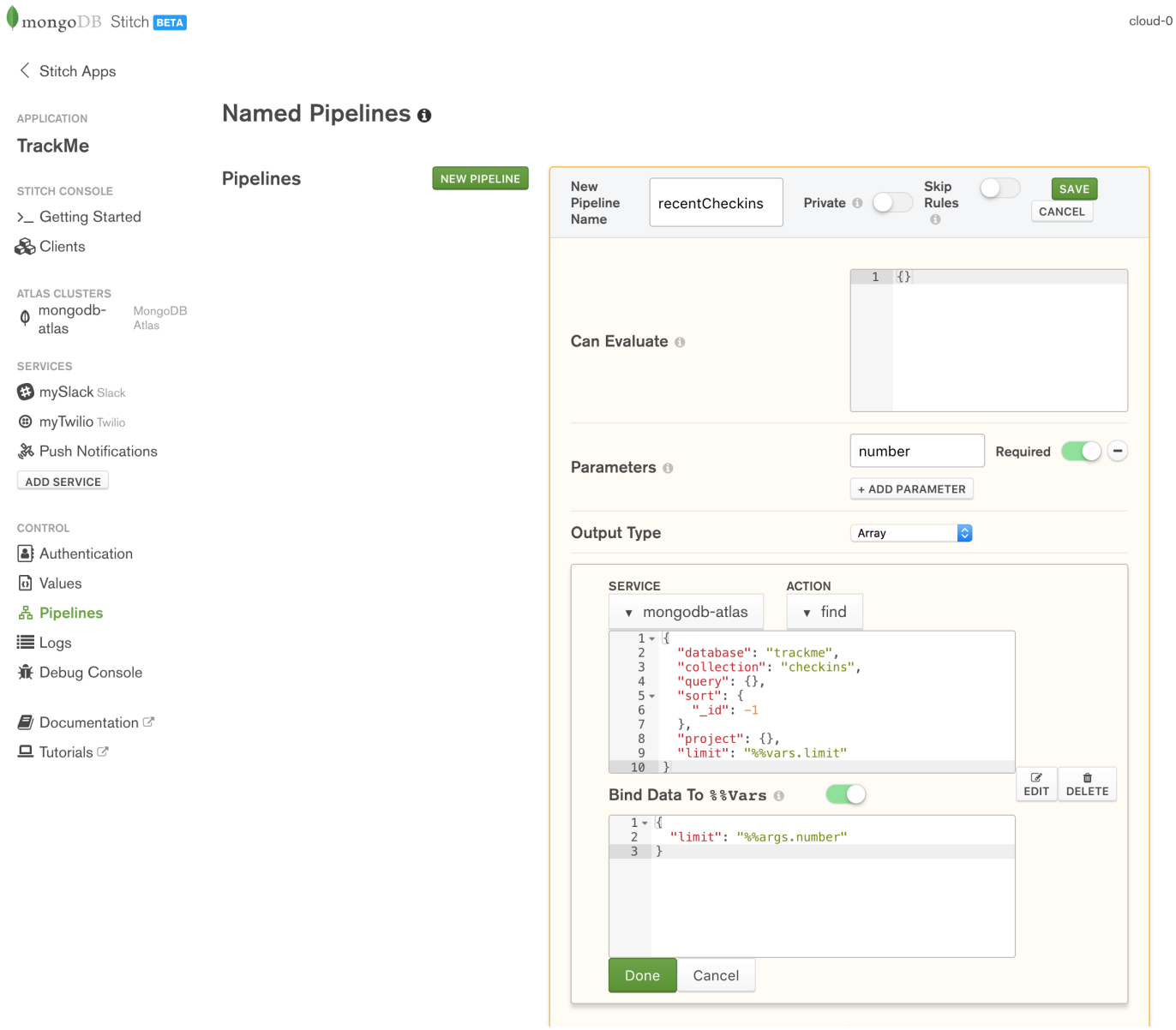
When creating a named pipeline, there is a set of information you must always provide:

- The **name** of the pipeline. The name is how your frontend application code, WebHooks, or other named pipelines can execute this pipeline.
- Whether the pipeline is **private**. If set to `true`, you can only invoke the pipeline from within the Stitch backend (from another named pipeline or a WebHook). If set to `false` then you can also invoke it directly from your

application's frontend code (or from Stitch's *Debug Console*).

- If a service accessed by your pipeline would otherwise be blocked by that resource's rules (e.g. a MongoDB document only being readable by the user that created it), enabling *Skip Rules* overrides those rules.
- You can control under what scenarios a pipeline is allowed to run by providing a JSON document – if it evaluates to `true` then the pipeline can run.
- You can define a set of **Parameters** that you can provide when invoking the pipeline. You can also tag as **Required**, those parameters which you must always provide.
- The **Output Type** indicates whether the pipeline will return a *Single Document*, *Boolean*, or *Array*.
- The rest of the pipeline definition consists of one or more stages, where each stage passes its results as input to the next. For each stage, you define:
  - Which **Service** to use (e.g. MongoDB, Twilio, Slack, or *built-in* (such as expressions, or literals))
  - The **Action** associated with that service (e.g. for a MongoDB service, you might pick `find` or `insert`)
  - The body of the action
  - **Bind Data to %%Vars** lets you create variables based on other values. When defining the value of one of these variables, you can use expansions such as:
    - **`%%args.parameter-name`** to access parameters passed to the pipeline
    - **`%%item.field-name`** to access the results of the previous stage
    - **`%%values.value-name`** to access pre-defined values/constants
  - You can access the variable values from the *Action* document using `%%vars.variable-name`.

The first pipeline to create is `recentCheckins` which returns an array of the user's most recent check-ins. When invoking the pipeline, the caller must provide a single parameter (`number`) which specifies how many check-ins it should return:



Note that the `trackme.checkins` collection already includes filters and rules to ensure that a user only sees their own check-ins and so the `query` sub-document can be empty.

Create the pipeline by pasting in the *Action* and *Bind Data To %%Vars* documents:

Action:

```
{
  "database": "trackme",
  "collection": "checkins",
  "query": {},
  "sort": {
    "_id": -1
  },
  "project": {},
  "limit": "%%vars.limit"
}
```

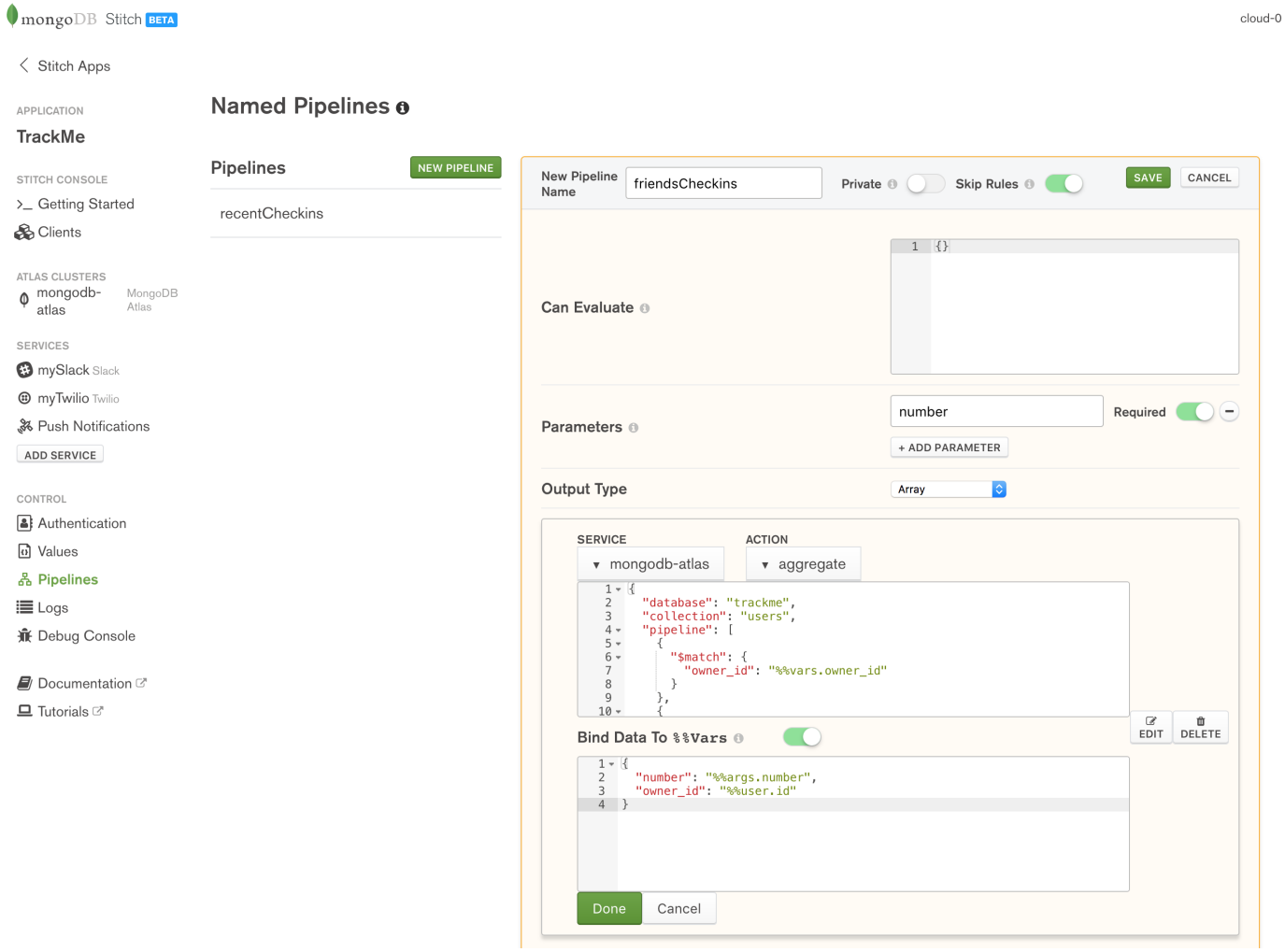
If you're not familiar with the MongoDB Query Language, this searches the `trackme.checkins` collection, reverse sorts on the `_id` (most recently inserted documents have the highest value), and then discards all but the first `%%vars.limit` documents.

Bind Data To %%Vars:

```
{
  "limit": "%%args.number"
}
```

This creates a `LET` statement where `%%vars.limit` is bound to the `number` parameter which the caller passes to the pipeline.

The second named pipeline to define is `friendsCheckins` to retrieve the most recent check-ins of users who have befriended the current user. Again, the caller must provide a parameter indicating the total number of check-ins it should return:



Create the pipeline by pasting in the *Action* and *Bind Data To %%Vars* documents:

Action:

```
{
  "database": "trackme",
  "collection": "users",
  "pipeline": [
    {
      "$match": {
        "owner_id": "%%vars.owner_id"
      }
    },
    {
      "$project": {
        "userData.email": 1,
        "_id": 0
      }
    },
    {
      "$lookup": {
        "from": "users",
        "localField": "userData.email",
        "foreignField": "friends",
        "as": "friendedMe"
      }
    },
    {
      "$project": {
        "friendedMe.owner_id": 1
      }
    },
    {
      "$unwind": {
        "path": "$friendedMe"
      }
    },
    {
      "$lookup": {
        "from": "checkins",
        "localField": "friendedMe.owner_id",
        "foreignField": "owner_id",
        "as": "friendsCheckins"
      }
    },
    {
      "$project": {
        "friendsCheckins": 1
      }
    },
    {
      "$unwind": {
        "path": "$friendsCheckins"
      }
    },
    {
      "$sort": {
        "friendsCheckins.id": -1
      }
    },
    {
      "$limit": "%%vars.number"
    },
    {
      "$group": {
        "id": "$friendsCheckins.email",
        "checkins": {
          "$push": {
            "venueName":
              "$friendsCheckins.venueName",
            "date": "$friendsCheckins.date",
            "url": "$friendsCheckins.url",
            "locationImg":
              "$friendsCheckins.locationImg"
          }
        }
      }
    }
  ]
}
```

If you're not familiar with the format of this action, take a look at the MongoDB Aggregation Framework.

Bind Data To %%Vars:

```
{
  "number": "%%args.number",
  "owner_id": "%%user.id"
}
```

As before, this makes the values passed in as parameters accessible to the pipeline's action section.

Before letting the user add a new email address to their array of friends, it's useful if you check that they aren't already friends:

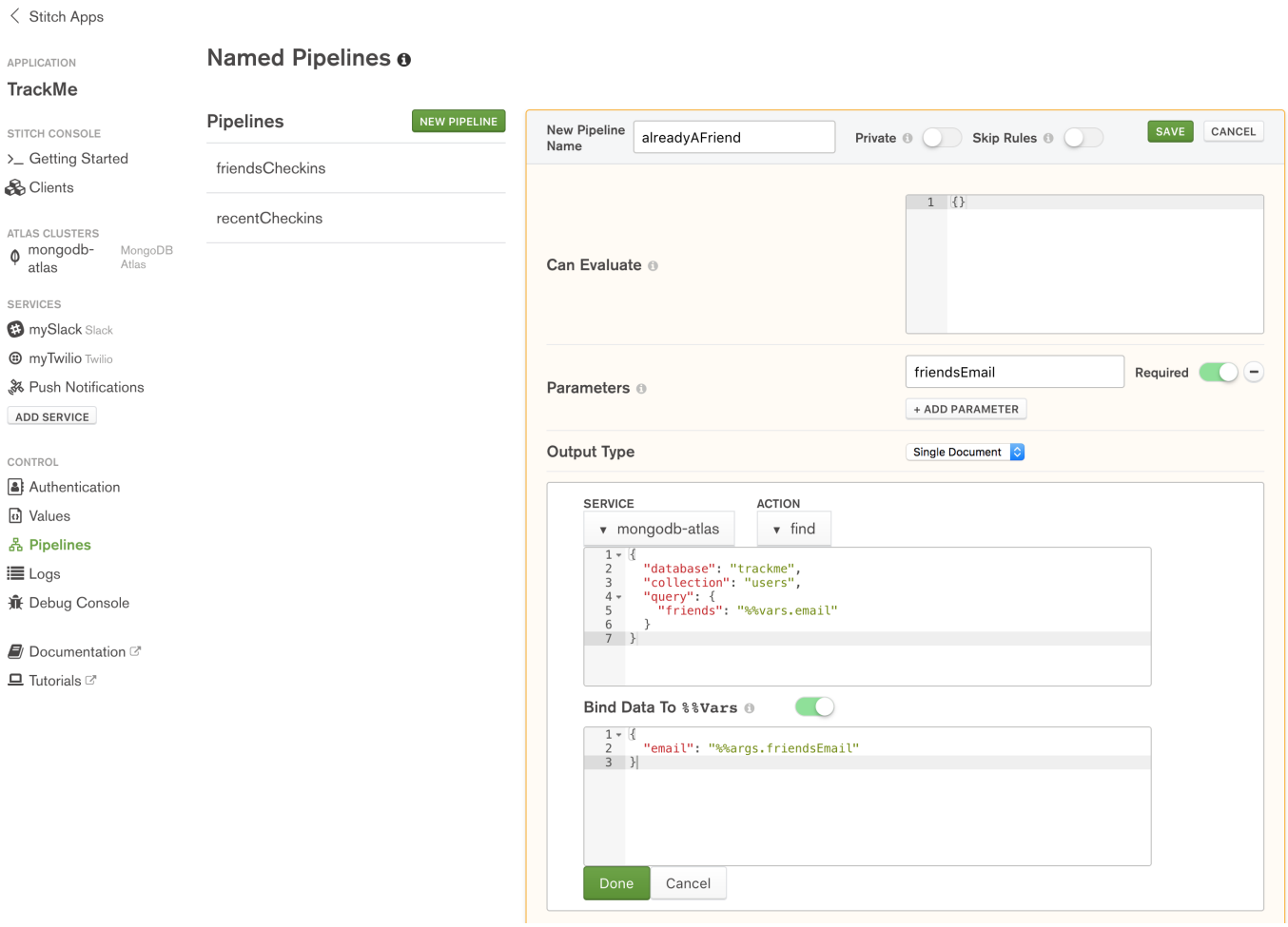
Action:

```
{
  "database": "trackme",
  "collection": "users",
  "query": {
    "friends": "%%vars.email"
  }
}
```

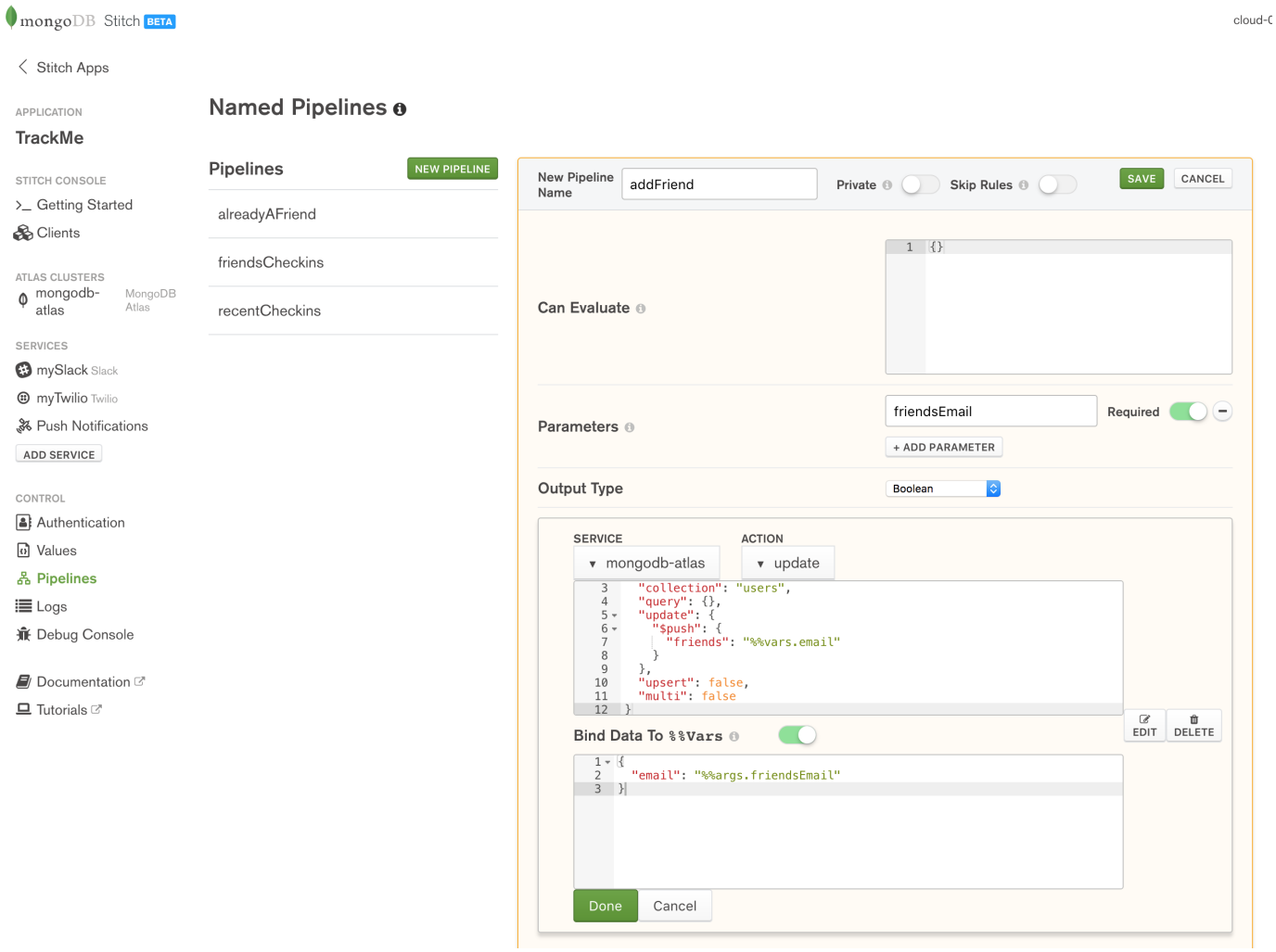
Because of the filter on the `trackme.users` collection, `find` operation will only look at this user, and so all the query needs to do is check if the provided email address already exists in the document's array of friends.

Bind Data To %%Vars:

```
{
  "email": "%%args.friendsEmail"
}
```



Once your application has checked that the requested friend isn't already listed, you can call the `addFrriend` pipeline to add their email address:



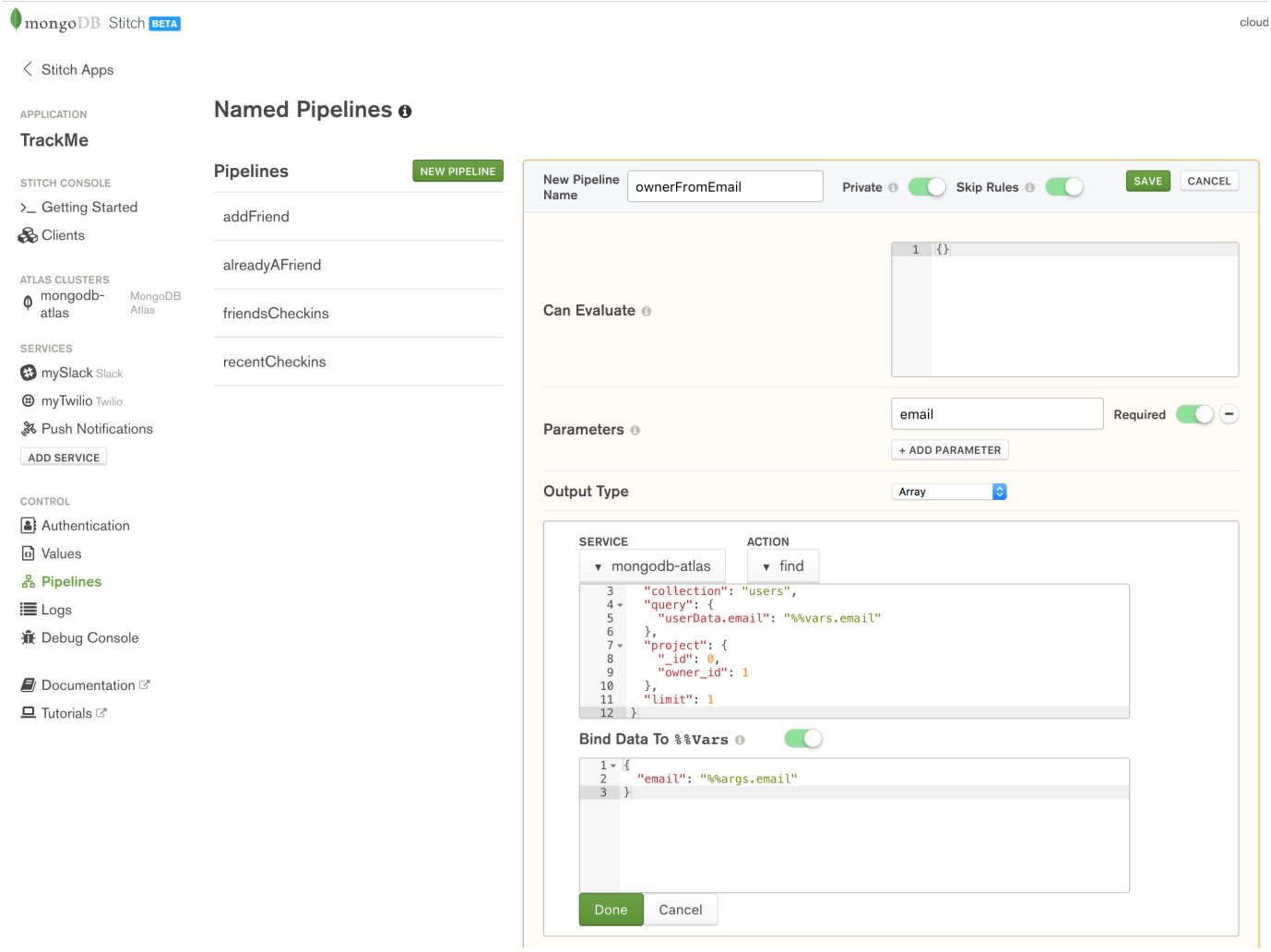
Action:

```
{
  "database": "trackme",
  "collection": "users",
  "query": {},
  "update": {
    "$push": {
      "friends": "%%vars.email"
    }
  },
  "upsert": false,
  "multi": false
}
```

Bind Data To %%Vars:

```
{
  "email": "%%args.friendsEmail"
}
```

When a user checks in through FourSquare or our iOS Workflow app, we identify them by their email address rather than their `owner_id`; the `ownerFromEmail` pipeline retrieves the user's `owner_id` using the `email` parameter:



Note that *Skip Rules* is enabled for the pipeline, so that it's able to search all documents in the `trackme.users` collection. For extra security, we make it *Private* so that it can only be executed by other pipelines or WebHooks that we create.

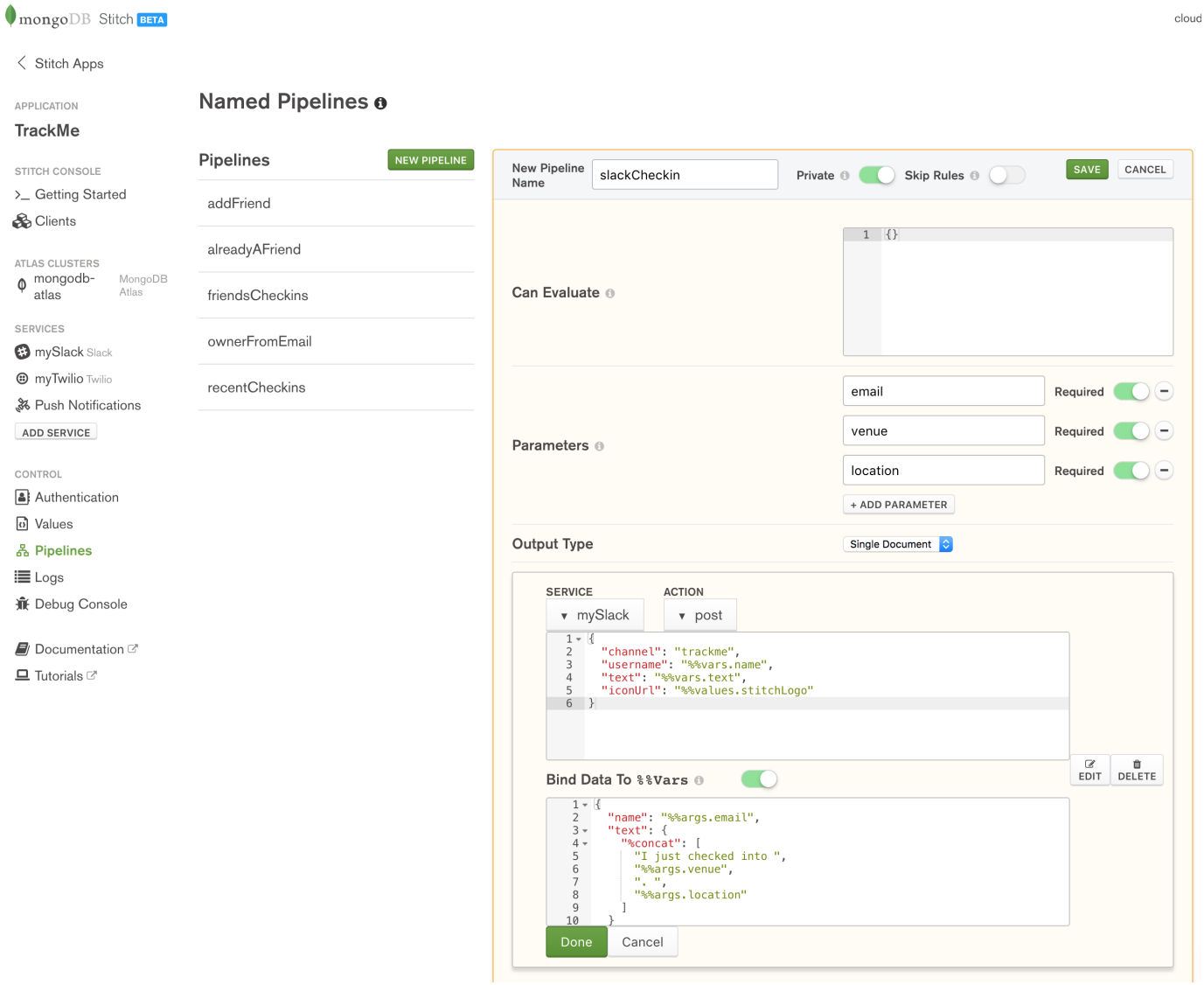
Action:

```
{
  "database": "trackme",
  "collection": "users",
  "query": {
    "userData.email": "%%vars.email"
  },
  "project": {
    "_id": 0,
    "owner_id": 1
  },
  "limit": 1
}
```

Bind Data To %%Vars:

```
{
  "email": "%%args.email"
}
```

When a user checks in, we want to send a notification to our Slack channel – create the `slackCheckin` pipeline to do so:



The pipeline uses the `mySlack` service that we created earlier. Again, set it to *Private* so that it can only be called from other WebHooks or named pipelines.

Action:

```
{
  "channel": "trackme",
  "username": "%%vars.name",
  "text": "%%vars.text",
  "iconUrl": "%%values.stitchLogo"
}
```

Bind Data To %%Vars:

```
{
  "name": "%%args.email",
  "text": {
    "%concat": [
      "I just checked into ",
      "%%args.venue",
      ". ",
      "%%args.location"
    ]
  }
}
```

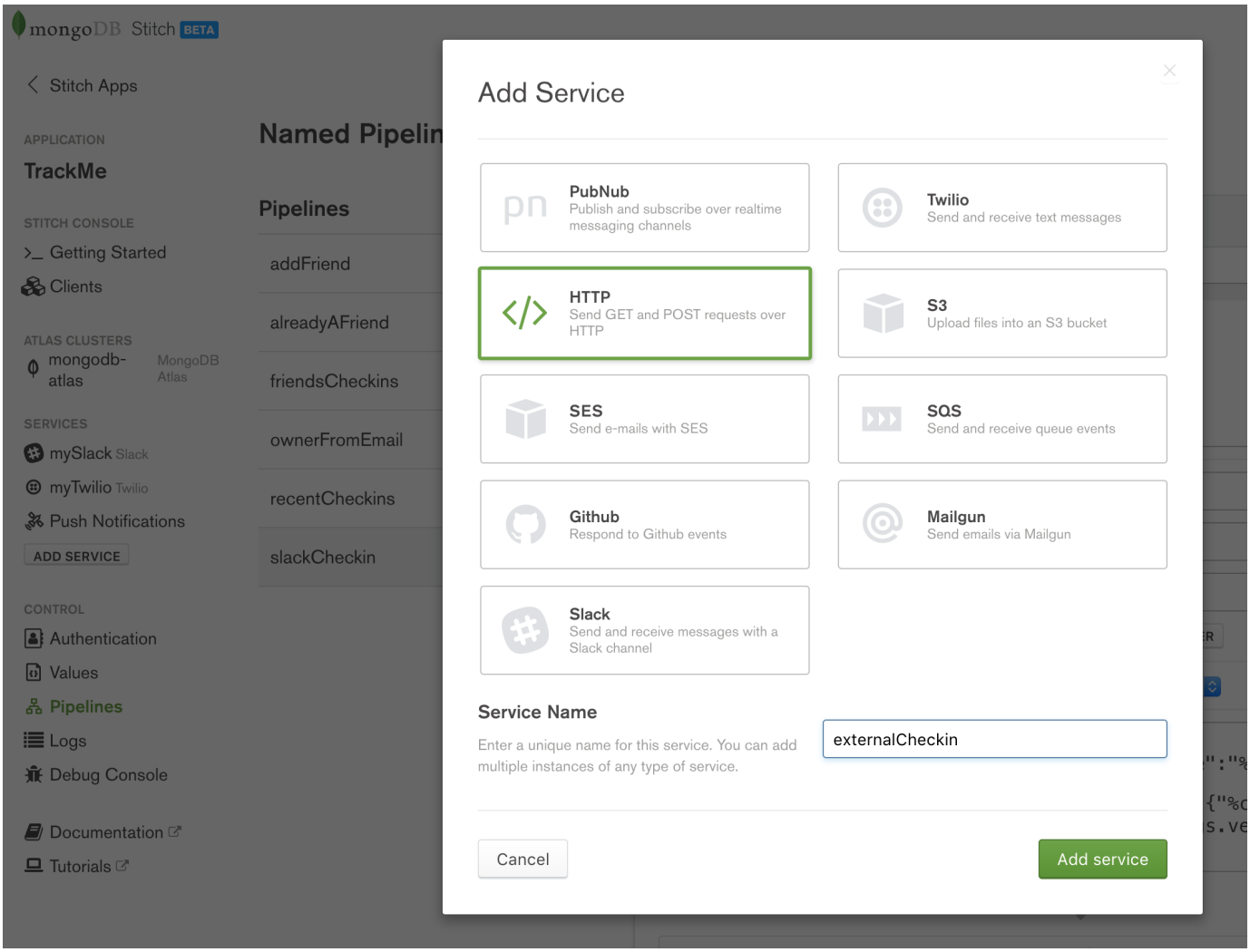
Working with other services – the HTTP service and WebHooks

The HTTP service fulfills two roles:

- Makes outgoing HTTP calls to services (either public web services or your microservices)
- Accepts incoming HTTP requests (through Stitch WebHooks) – allowing external services to trigger actions within your Stitch application

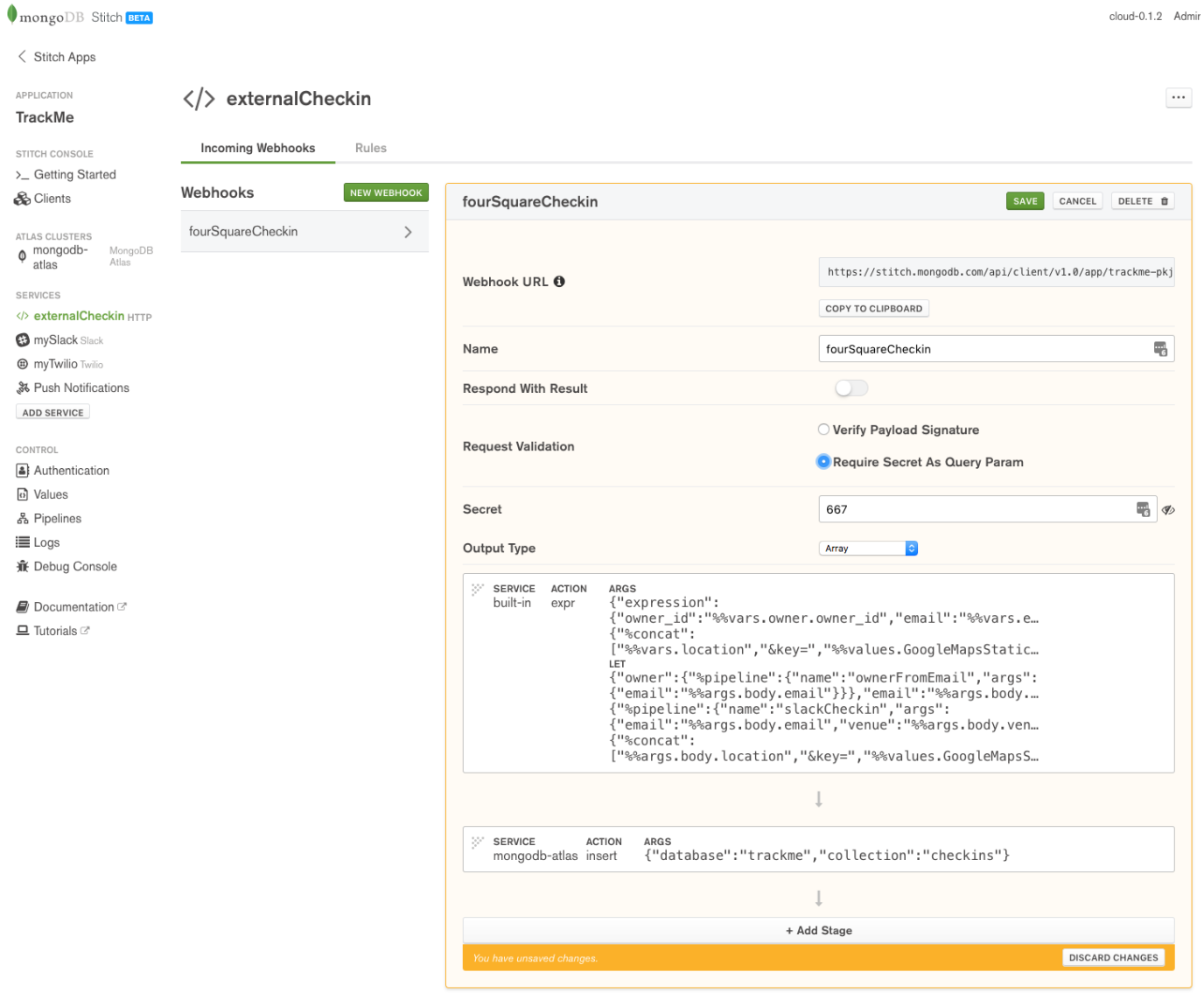
The TrackMe application uses WebHooks to receive notifications whenever one of our users checks in through FourSquare or the iOS Workflow app.

Create a new HTTP service called `externalCheckin`:



There's no need to define any (outgoing) rules as our application doesn't use this service to send out any requests.

Create the `fourSquareCheckin` WebHook:



To prevent another application sending your application bogus check-ins, enable *Require Secret As Query Param* and provide a secret (I've used `667`, but for a production app, you'd want a stronger secret).

The WebHook consists of two stages. The first stage (Stage 0) uses the *built-in expression* action to build a JSON document containing the check-in data. Note that we form the `locationImg` field by adding our `GoogleMapsStaticKey` value to the end of the received URL (so that the new URL can be used by the frontend application code to retrieve the map image from Google).

Action (first stage):

```
{
  "expression": {
    "owner_id": "%%vars.owner.owner_id",
    "email": "%%vars.email",
    "venueName": "%%vars.venue",
    "date": "%%vars.date",
    "url": "%%vars.url",
    "locationImg": {
      "%concat": [
        "%%vars.location",
        "&key=",
        "%%values.GoogleMapsStaticKey"
      ]
    }
  }
}
```

When creating the variables to construct the expression, `%%vars.owner` is formed by invoking our `ownerFromEmail` named pipeline – passing in the received `email` address from the received HTTP body.

Bind Data To %%Vars (first stage):

```
{
  "owner": {
    "%pipeline": {
      "name": "ownerFromEmail",
      "args": {
        "email": "%%args.body.email"
      }
    }
  },
  "email": "%%args.body.email",
  "venue": "%%args.body.venue",
  "date": "%%args.body.checkinDate",
  "url": "%%args.body.url",
  "location": "%%args.body.location",
  "slackDummy": {
    "%pipeline": {
      "name": "slackCheckin",
      "args": {
        "email": "%%args.body.email",
        "venue": "%%args.body.venue",
        "location": {
          "%concat": [
            "%%args.body.location",
            "&key=",
            "%%values.GoogleMapsStaticKey"
          ]
        }
      }
    }
  }
}
```

When defining the variables, we also create a dummy variable (`slackDummy`) so that we can invoke the `slackCheckin` pipeline as a side effect.

The second stage takes that document and stores it in the `trackme.checkins` collection.

Action (second stage):

```
{
  "database": "trackme",
  "collection": "checkins"
}
```

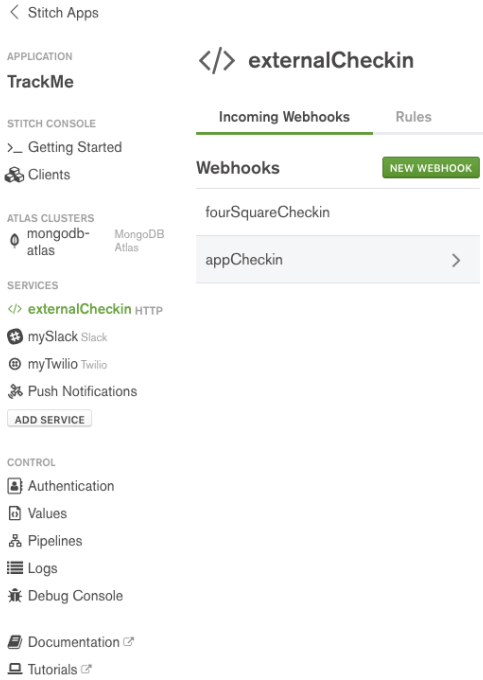
Take a note of the WebHook URL (`https://stitch.mongodb.com/api/client/v1.0/app/trackme-pkjif/svc/externalCheckin/incomingWebhook/598081f44fdd1f5eb7900c16` in this example) as this is where other services must send requests.

The second WebHook (`appCheckin`) will be invoked from the iOS Workflow app; it's very similar to `fourSquareCheckin` but there's no need to add the Google Maps key as for these check-ins, `locationImg` is the Imgur URL of a photo taken by the user at the venue.

Action (first stage):

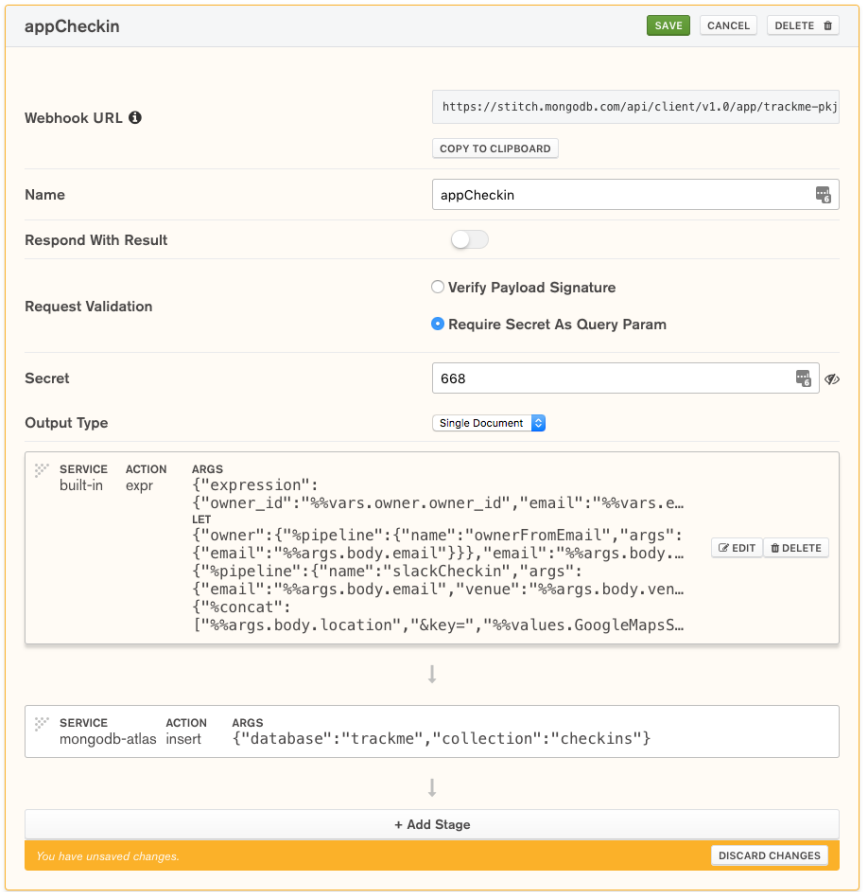
```
{
  "expression": {
    "owner_id": "%%vars.owner.owner_id",
    "email": "%%vars.email",
    "venueName": "%%vars.venue",
    "date": "%%vars.date",
    "url": "%%vars.url",
    "locationImg": "%%vars.location"
  }
}
```

Bind Data To %%Vars (first stage):




```
{
  "owner": {
    "%pipeline": {
      "name": "ownerFromEmail",
      "args": {
        "email": "%%args.body.email"
      }
    }
  },
  "email": "%%args.body.email",
  "venue": "%%args.body.venue",
  "date": "%%args.body.date",
  "url": "%%args.body.url",
  "location": "%%args.body.location",
  "slackDummy": {
    "%pipeline": {
      "name": "slackCheckin",
      "args": {
        "email": "%%args.body.email",
        "venue": "%%args.body.venue",
        "location": {
          "%concat": [
            "%%args.body.location",
            "&key=",
            "%%values.GoogleMapsStaticKey"
          ]
} } } } }
```

WebHook definition (second stage):



```
{
  "database": "trackme",
  "collection": "checkins"
}
```

Take a note for the Webhook URL.

Checking into the app using WebHooks

Capturing FourSquare check-ins (via IFTTT)

IFTTT (If This Then That) is a free cloud service which allows you to automate tasks by combining existing services (Google Docs, Facebook, Instagram, Hue lights, Nest thermostats, GitHub, Trello, Dropbox,...). The name of the service comes from the simple pattern used for each Applet (automation rule): "**IF T**his event occurs in service x **T**hen trigger **T**hat action in service y".

IFTTT includes a Maker service which can handle web requests (triggers) or send web requests (actions). In this case, you can create an Applet to invoke our `fourSquareCheckin` WebHook whenever you check in using the Swarm (Foursquare) app:

Note that you form the URL: (`https://stitch.mongodb.com/api/client/v1.0/app/trackme-pkjif/svc/externalCheckin/incomingWebhook/598081f44fdd1f5eb7900c16?secret=667`) from the WebHook URL, with the addition of the `secret` parameter.

The HTTP method is set to POST and the body is a JSON document formed from several variables provided by the FourSquare service:

```
{
    "email":"me@gmail.com",
    "venue":"{{VenueName}}",
    "checkinDate":"{{CheckinDate}}",
    "url":"{{VenueUrl}}",
    "location":"{{VenueMapImageUrl}}"
}
```

In this example, the email is hard-coded, and so all check-ins will be registered by the same user. A production application would need a better solution.

Checking in from an iPhone (via the Workflow iOS app)

iOS Workflow has some similarities with IFTTT, but there are also some significant differences:

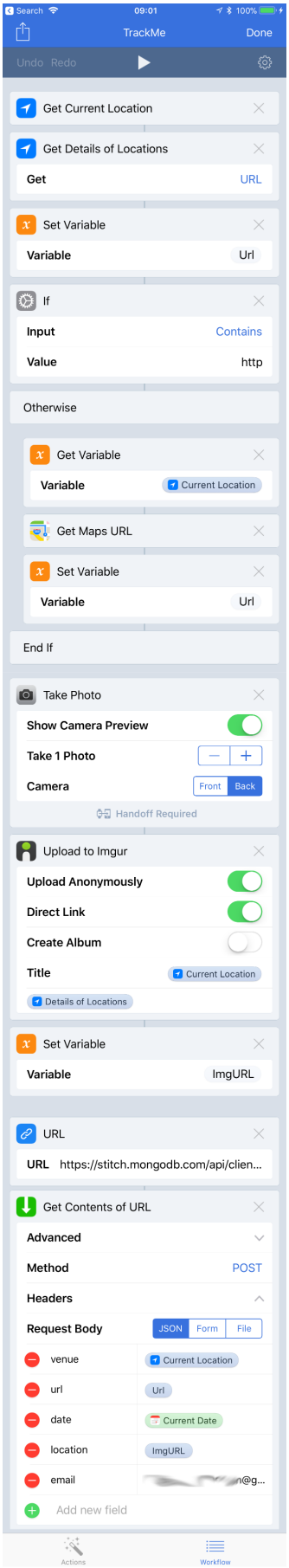
- Workflow runs on your iOS device rather than in the cloud.
- You trigger Workflows by performing actions on your iOS device (e.g. pressing a button); external events from cloud service trigger IFTTT actions.
- Workflow allows much more involved patterns than IFTTT; it can loop, invoke multiple services, perform calculations, access local resources (e.g. camera and location information) on your device, and much more.

Implementing a Workflow involves dragging actions into the work area and then adding attributes to those actions (such as the URL for the TrackMe `appCheckin` WebHook). The result of one action is automatically used as the input to the next in the workflow. Results can also be stored in variables for use by later actions.

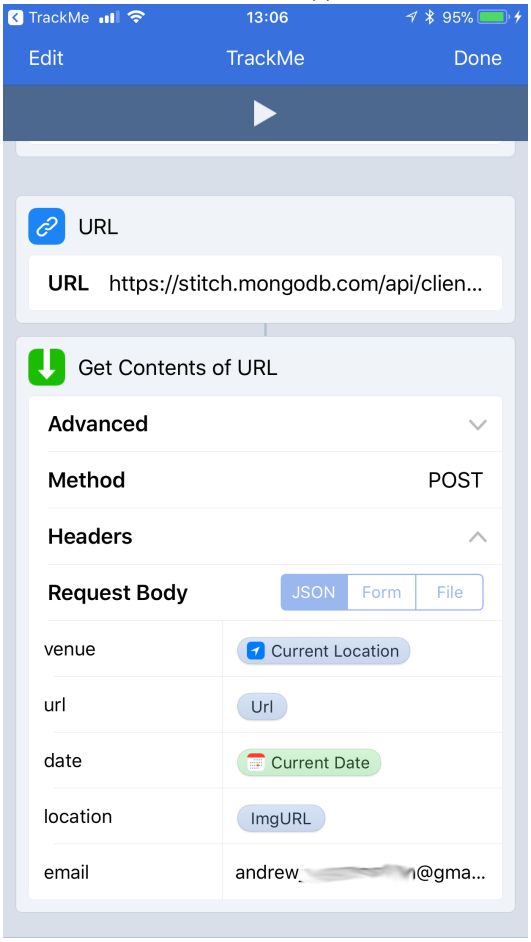
The TrackMe workflow:

- Retrieve the current location from your device & fetch details venue details
- If the venue details isn't a URL then fetch an Apple Maps URL
- Take a new photo and upload it to Imgur
- Create a URL to invoke Trackme (ending in `?secret=668`)
- Perform an HTTP POST to this URL, including check-in details in the body

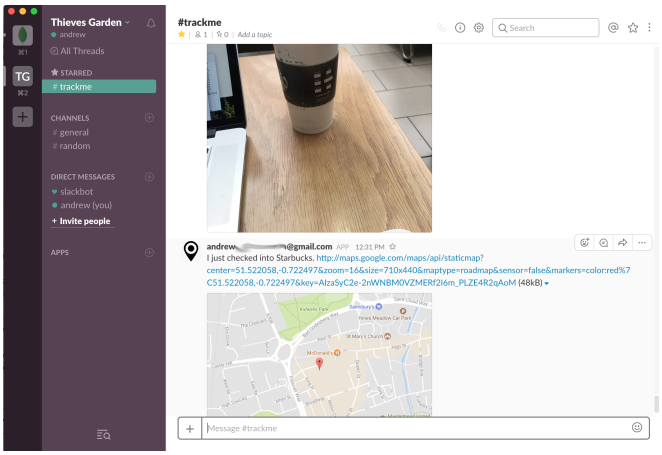
This is the Check In workflow:



You can see the Workflow applet in action here:



Checking the *trackme* Slack channel confirms that the checkin was received. Note that you also check the results of the request in the *Logs* section of the Stitch Admin UI.



Building a frontend app using React

In an earlier chapter, I covered developing an application frontend using React. In this post, I don't rehash the

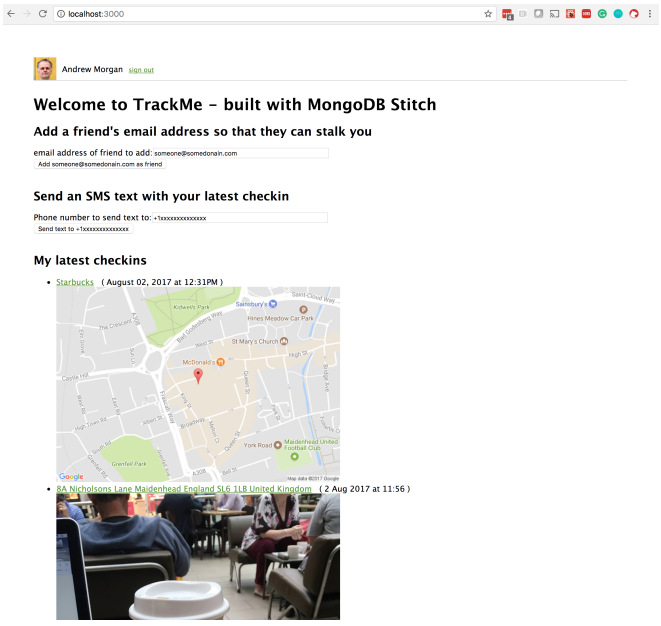
introduction to React, instead, I focus on how the React application interacts with its Stitch backend.

You may recall that the earlier React application frontend included writing a data service to handle interactions with the backend Mongopop REST API; this isn't required for the new TrackMe frontend as the Stitch SDK provides our access to the backend.

The TrackMe application frontend allows a user to:

- Log in using Google or Facebook authentication
- View their most recent check-ins
- View the most recent check-ins of users that have added them to their list of friends
- Add another user to their list of friends
- Use Twilio to send an SMS text to any number, containing the user's latest check-in information

Download the full application can from the *trackme_MongoDB_Stitch* GitHub project.



To run the TrackMe frontend:

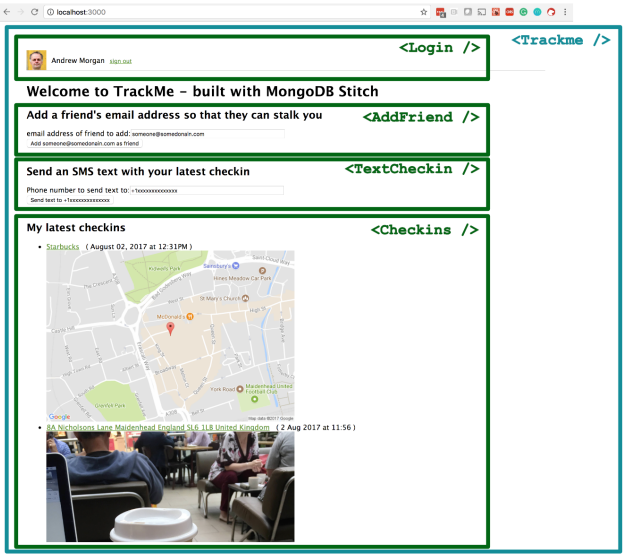
```
git clone https://github.com/am-MongoDB/\
  trackme_MongoDB_Stitch.git
cd trackme_MongoDB_Stitch
npm install
```

Edit the value of `appId` in `src/config.js`; replacing `trackme-xxxx` with the value for your Stitch app (found in the *Clients* tab in the Stitch console after creating your MongoDB Stitch app).

```
npm start
```

ReactJS JavaScript (ES6) client code

The application's React frontend is made up of the *Trackme* component which embeds four sub-components:



Any Stitch JavaScript application must start by importing the Stitch SKD `StitchClient`. The code then uses `StitchClient` to connect to MongoDB Stitch in the *Trackme* component's constructor function within `App.js`. After instantiating `stitchClient`, it's used to connect to the `trackme` database, followed by the `checkins`, and `user` collections:

```
import { StitchClient } from 'mongodb-stitch';
import config from './config';
...
class Trackme extends React.Component {
  constructor(props) {
    super(props);
        ...
    this.appId = config.appId;
        ...
    let options = {};

    this.stitchClient = new StitchClient(
      this.appId, options);

    this.db = this.stitchClient.service(
      "mongodb", "mongodb-atlas")
        .db("trackme");

    this.checkins = this.db.collection(
      "checkins");
    this.users = this.db.collection("users");
    }
...
}
```

`stitchClient` is passed down to `src/login.component.js`, where a single line of code can be used to authenticate using Facebook:

```
<div
  onClick={() =>
    this.props.stitchClient.authWithOAuth(
      "facebook")}
  className="signin-button">
  <div className="facebook-signin-logo" />
  <span className="signin-button-text">
    Sign in with Facebook
  </span>
</div>
```

The same component can use Google to authenticate in the same way:

```
<div
  onClick={() =>
    this.props.stitchClient.authWithOAuth(
      "google")}
  className="signin-button"
>
  ...
  <span className="signin-button-text">
    Sign in with Google
  </span>
</div>
```

Whichever authentication method was used, common code displays the user's name and avatar (in `src/login.component.js`):

```
this.props.stitchClient.userProfile()
.then(
  userData => {
    ...
    this.setState({userData: userData.data});
    ...
  },
  error => {
    // User hasn't authenticated yet
  })
...
{this.state.userData &&
    this.state.userData.picture
  ? <img src={this.state.userData.picture}
    className="profile-pic" alt="mug shot"/>
  : null}
<span className="login-text">
  <span className="username">
    {this.state.userData &&
      this.state.userData.name
      ? this.state.userData.name
      : "?"}
  </span>
</span>
```

Common code can logout the user (in `src/login.component.js`):

```
this.props.stitchClient.logout()
```

Once logged in, the application frontend can start making use of the services that we've configured for this app through the Stitch UI. In this case, we directly insert or update the user's details in the `trackme.users` collection (in `src/login.component.js`):

```
this.props.userCollection.updateOne(
  {},  /* We don't need to identify the
        user in the query as the
        pipeline's filter will handle
        that.*/
  {
    $set: {
      owner_id:
      this.props.stitchClient.authedId(),
      userData: userData.data
    }
  },
  {upsert: true})
.then(
  result=>{},
  error=>{console.log("Error: " + error)}
  );
```

While that code is using the Stitch SDK/API, it is invoking the MongoDB Atlas service in a traditional manner by performing an `updateOne` operation **but** the Stitch filters and rules we've configured for the `users` collection will still be enforced.

In this React application frontend, I have intentionally used a variety of different ways to interact with Stitch – you will later see how to call a named pipeline and how to construct and execute a new pipeline on the fly.

When adding a new friend, two of the named pipelines we created through the Stitch UI (`alreadyAFriend` & `addFriend`) are executed to add a new email address to the list if and only if it isn't already there (`src/addfriend.component.js`):

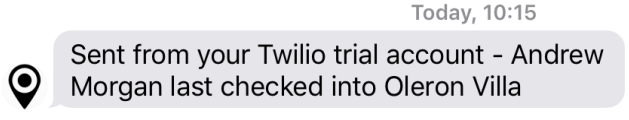
```
import { builtins } from 'mongodb-stitch';
...
this.props.stitchClient
  .executePipeline([
    builtins.namedPipeline('alreadyAFriend',
      {friendsEmail: email})])
  .then(
    response => {
      if (response.result[0]) {
        this.setState({error: email +
          " has already been included as a friend."});
      } else
      {
        this.props.stitchClient
          .executePipeline([
            builtins.namedPipeline(
              'addFriend', {friendsEmail: email})])
          .then(
            response => {
              if (response.result[0]) {
                this.setState({success:
                  email + " added as a friend" they can now\
                  see your checkins."});
              } else {
                this.setState({
                  error: "Failed to add "
                    + email
                    + " as a friend"});
              }
            },
            error => {
              this.setState({error: "Error: "
                + error});
              console.log({error: "Error: "
                + error});
            }
          )
      }
    },
    error => {
      this.setState({error: "Error: "
        + error});
      console.log({error: "Error: "
        + error});
    }
  )
...
```

`src/text.checkin.component.js` finds the latest checkin (for this user), and then creates and executes a new service pipeline on the fly – sending the venue name to the requested phone number via Twilio:

```
this.props.checkins.find({},
  {sort: {_id: -1}, limit: 1})
.then (
  response => {
    venue = response[0].venueName;
    this.props.stitchClient.userProfile()
    .then (
      response => {
        name = response.data.name;
      })
    .then (
      response => {

        this.props.stitchClient
          .executePipeline([
            {
              service: "myTwilio",
              action: "send",
              args: {
                to: this.state.textNumber,
                from: "%%values.twilioNumber",
                body: name
                  + " last cheked into "
                  + venue
              }
            }
          ])
          .then(
            response => {
              this.setState({success:
                "Text has been sent to "
                + this.state.textNumber});
            },
            error => {
              this.setState({error:
                "Failed to send text: "
                  + error});
              console.log({error:
                "Failed to send text: "
                  + error});
            })
      })
  },
    error => {
    this.setState({error:
      "Failed to read the latest checkin: "
        + error});
    }
  )
```

Note that the pipeline refers to `%%values.twilioNumber` – this is why that value couldn't be tagged as *Private* within the Stitch UI.

This is the result:



The checkins for the user and their friends are displayed in the `Checkins` component in `src/checkins.component.js`. The following code invokes the `recentCheckins` named pipeline (including the

`number` parameter to request the 10 most recent checkins):

```
this.props.stitchClient.executePipeline([
    builtins.namedPipeline('recentCheckins',
      {number: 10})])
  .then(
    checkinData => {
      this.setState({checkins: checkinData
        .result[0]
        .map((checkin, index) =>
        <li key={index}>
          <a href={checkin.url}
          target="_Blank">{checkin.venueName}
          </a>
          ( {checkin.date} )
           <br/>
           <img src={checkin.locationImg}
           className="mapImg"
          alt={"map of " + checkin.venueName}
          />
        </li>
      )})
    },
    error => {
      console.log(
        "Failed to fetch checkin data: "
          + error)
  })
```

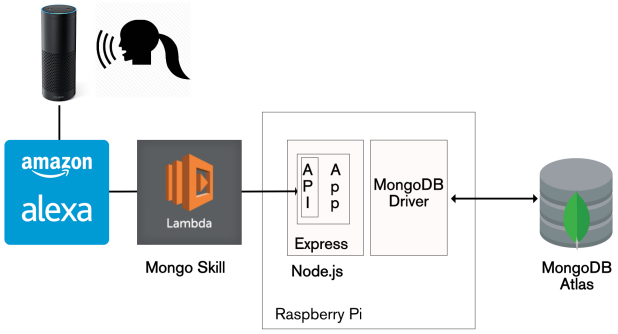
A similar code block executes the `friendsCheckin` named pipeline and then loops over each of the friends, displaying the latest checkins for each one:

```
this.props.stitchClient
  .executePipeline([
    builtins.namedPipeline('friendsCheckins',
      {number: 10})])
  .then(
    friendData => {
      this.setState({friendsCheckins:
        friendData.result[0].map((friend,
          friendIndex) =>
          <li key={friendIndex}>
            <strong>{friend._id}</strong>
            <ul>
              {friend.checkins.map((checkin)
                =>
                <li>
                  <a href={checkin.url}
                    target="_Blank">
                    {checkin.venueName}</a>
                     ( {checkin.date} )
                      <br/>
                    <img
                     src={checkin.locationImg}
                     className="mapImg"
                    alt={"map of "
                      + checkin.venueName}/>
                </li>
              )}
            </ul>
          </li>
        )
      })
    },
    error => {
      console.log(
        "Failed to fetch friends' data: "
          + error)
  })
```

Continue accessing your data from existing applications (Amazon Alexa skill)

Not every MongoDB Stitch use-case involves building a greenfield app on top of a brand new data set. It's common that you already have a critical application, storing data in MongoDB, and you want to safely allow new apps or features to use some of that same data.

The good news is that your existing application can continue without any changes, and Stitch can be added to control access from any new applications. To illustrate this, you can reuse the *Mongo* Alexa Skill created in the previous chapter. The JavaScript code needs a sight adjustment (due to a change I made to the schema) – use `alexa/index.js`.



The Alexa skill uses the Express/Node.js REST API implemented earlier, illustrating how existing application can continue to work with MongoDB data after introducing Stitch.

Chapter summary

MongoDB Stitch lets you focus on building applications rather than on managing data manipulation code, service integration, or backend infrastructure. Whether you're just starting up and want a fully managed backend as a service, or you're part of an enterprise and want to expose existing MongoDB data to new applications, Stitch lets you focus on building the app users want, not on writing boilerplate backend logic.

In this chapter, you've learned how to:

- Create a new MongoDB Stitch app that lets you access data stored in MongoDB Atlas
- Integrate with authentication providers such as Google and Facebook
- Configure data access controls – ensuring that application end-users can access just the information they're entitled to
- Enable access to the Slack, and Twilio services
- Define constants/values that you can use securely within your application backend, without exposing them to your frontend code, or the outside world
- Implement named pipelines to access MongoDB and your other services
- Implement WebHooks that allow external services to trigger events in your application
- Invoke your new WebHooks from other applications
- Implement your application frontend in React/ JavaScript
  - Authenticate users using Google and Facebook
  - Use the MongoDB Query Language to "directly" access your data through Stitch
  - Execute names pipelines
  - Create and run new pipelines on the fly
- Continue to access the same MongoDB data from existing apps, using the MongoDB drivers

When reviewing the earlier chapters, it's interesting to assess what work you could avoid:

- 1: Introducing The MEAN Stack (and the young MERN upstart)
  - This is an introductory chapter, much of it is still relevant when using Stitch.
- 2: Using MongoDB With Node.js
  - The work in this chapter isn't needed as Stitch works with MongoDB Atlas directly, removing the need to use the MongoDB Driver for Node.JS
- 3: Building a REST API with Express.js
  - This chapter isn't needed as Stitch provides the SDK/REST API to access your MongoDB data (and other services).
- 4: Building a Client UI Using Angular 2 (formerly AngularJS) & TypeScript
  - Much of this chapter is still relevant if you're looking to build your application frontend using Angular2, but the work to implement the data service isn't needed when working with MongoDB Stitch.
- 5: Using ReactJS, ES6 & JSX to Build a UI (the rise of MERN)
  - Much of this chapter is still relevant if you're looking to code your application frontend using React, but the work to implement the data service isn't needed when working with MongoDB Stitch.
- 6: Browsers Aren't the Only UI – Mobile Apps, Amazon Alexa, Cloud Services
  - Covers much of the same ground as this post. Some of the examples can be simplified when using MongoDB Stitch, and the extending of the REST API

to add new features isn't necessary when using Stitch.

The following table summarizes the steps required to build an application without the help of Stitch:

	Without Stitch	With Stitch
Configure MongoDB Cluster	Use Atlas	Use Atlas
Configure Stitch Services Pipeline, Rules, and WebHooks	No	Yes
Code user authentication	Yes	No
Code data access controls	Yes	No
Code against Twilio, and Slack APIs	Yes	No
Provision Node.js backend to service static content	Yes	Yes
Provision Node.js backend to interact with MongoDB and all other services	Yes	No
Code REST API	Yes	No
Code UI in React frontend	Yes	Yes
Code data access service in frontend (to use REST API)	Yes	No
Code business logic in frontend	Yes	Yes
Make REST call from iOS Workflow app, IFTTT, & Alexa Skill	Yes	Yes

Both MongoDB Atlas and MongoDB Stitch come with a free tier – so go ahead and try it out for yourself.

We Can Help

We are the MongoDB experts. Over 4,300 organizations rely on our commercial products, including startups and more than half of the Fortune 100. We offer software and services to make your life easier:

MongoDB Enterprise Advanced is the best way to run MongoDB in your data center. It's a finely-tuned package of advanced software, support, certifications, and other services designed for the way you do business.

MongoDB Atlas is a database as a service for MongoDB, letting you focus on apps instead of ops. With MongoDB Atlas, you only pay for what you use with a convenient hourly billing model. With the click of a button, you can scale up and down when you need to, with no downtime, full security, and high performance.

MongoDB Stitch is a backend as a service (BaaS), giving developers full access to MongoDB, declarative read/write controls, and integration with their choice of services.

MongoDB Cloud Manager is a cloud-based tool that helps you manage MongoDB on your own infrastructure. With automated provisioning, fine-grained monitoring, and continuous backups, you get a full management suite that reduces operational overhead, while maintaining full control over your databases.

MongoDB Professional helps you manage your deployment and keep it running smoothly. It includes support from MongoDB engineers, as well as access to MongoDB Cloud Manager.

Development Support helps you get up and running quickly. It gives you a complete package of software and services for the early stages of your project.

MongoDB Consulting packages get you to production faster, help you tune performance in production, help you scale, and free you up to focus on your next release.

MongoDB Training helps you become a MongoDB expert, from design to operating mission-critical systems at scale. Whether you're a developer, DBA, or architect, we can make you better at MongoDB.

Resources

For more information, please visit mongodb.com or contact us at sales@mongodb.com.

Case Studies (mongodb.com/customers)
Presentations (mongodb.com/presentations)
Free Online Training (university.mongodb.com)
Webinars and Events (mongodb.com/events)
Documentation (docs.mongodb.com)
MongoDB Enterprise Download (mongodb.com/download)
MongoDB Atlas database as a service for MongoDB (mongodb.com/cloud)
MongoDB Stitch backend as a service (mongodb.com/cloud/stitch)

US 866-237-8815 ▪ INTL +1-650-440-4474 ▪ info@mongodb.com