TicTacToe Documentation

Release 1

Chris Horuk

Sep 25, 2017

Contents

Contents:

CHAPTER 1

Introduction

This documentation is in support of the TicTacToe application, which consists of a Google App Engine backend written in Scala and Java and an iOS client application. The documentation assumes that you are already familiar with using Google App Engine and Apache Ant with Scala and Java. If you need a refresher, take a look at the documentation and code for my previous tutorial on these topics.

The intent of this documentation is to provide an in-depth explanation of how you can use Google App Engine to host a Scala backend that uses the Google Cloud Endpoints service to provide a uniform set of APIs accessible from web, Android and iOS clients. The sample application itself only implements the iOS client, but the Scala backend can support Android or web clients as well. Another intent of the documentation is to discuss the use of the Objectify framework to interact with the AppEngine datastore and address any caveats with its interaction with the Google Cloud Endpoints service.

The documentation is structured to first introduce you to the various technologies used by the TicTacToe application along with any configurations necessary to use them within TicTacToe. You are then provided with a high level description of the implementation of the complete TicTacToe application. Finally, there are multiple sections on the application code, one for each major portion of the code, that further explain any required configurations and encountered limitations.

CHAPTER 2

Technologies

The complete TicTacToe application incorporates a number of different technologies and services into its implementation. In this section, I will provide a quick overview of each technology, discuss why it was needed for the TicTacToe application and explain any extra configuration that was needed to successfully integrate it within the application.

Google App Engine

As already stated in the introduction, this documentation assumes prior knowledge on the part of the reader. Specifically, this documentation assumes you are familiar with using Scala, Java and Ant to develop and deploy applications for Google App Engine. If you need a refresher, take a look at the documentation and code for my previous tutorial on these topics. One point of interest not covered in my documentation is the use of the administration console when developing and testing your application. For every application that you host on Google App Engine, you can access an admin console that will let you monitor all kinds of information about that application, perform various administrative tasks and configure billing options. While developing and testing the API backend and iOS client of the TicTacToe application, I used two sections of the admin console extensively once I realized how to use them: the application logs and the datastore viewer. If you upload your application to App Engine, you can use the application logs to ensure your requests are completing successfully, as well as monitor any log messages you are outputting, and the datastore viewer for an accurate representation of the contents of your application's datastore.

Google Cloud Endpoints

Google Cloud Endpoints is a relatively new technology introduced by Google for their Google App Engine platform-as-a-service. The endpoints technology allows App Engine developers to generate APIs and client libraries from an annotated App Engine application using the endpoints tool included with the SDK. The annotated App Engine application is referred to as an API backend. The beauty of Google Cloud Endpoints is that these exposed APIs can all be implemented using the various services provided by App Engine itself and the API developers never have to worry about load balancing, scaling and so on. With an App Engine API backend, you can develop a shared web backend that can be used to provide a uniform experience across multiple different client applications. The endpoints tool, located in the App Engine Java SDK bin directory, comes in two flavors: a *.sh* for Linux and OS X and a *.cmd* for Windows. For the rest of this documentation, when I refer to the endpoints tool, I am referring to *endpoints.sh*.

Developing an API Backend

The development process to create an API backend is fairly straightforward and starts off, as you would expect, with implementing the APIs that you want to expose. While developing your APIs, keep in mind that the parameters your APIs use and the objects they return will be provided and accessed by client applications that are likely written in a completely different language. As such, there are some restrictions on the types allowed. For parameter types, only basic types like string and int are allowed, as well as Plain Old Java Objects (POJOs). This means that you cannot pass a Collection object as a parameter unless you have written a wrapper around it to turn it into a POJO. The restrictions on the return type are almost the opposite, you can only return POJOs, arrays and Collections; nothing with a simple type. This is mostly due to how these objects are serialized when they are transferred in between the API backend and the client application.

Once you have some APIs that you want to expose, the next step is to properly annotate them. Google's documentation introduces the necessary annotations and provides a detailed description of how to properly annotate your APIs, although there are a few caveats that are not easily discernable from their docs. One such caveat is related to the *@ApiMethod* annotation options, specifically the path option. The path configuration is optional and if left blank, the endpoints tool will just pick a default path based on the name of the API. However, if you have multiple APIs with the same underlying HTTP method (for example a GET request that returns one entity and a GET request that returns a list of entities) and you leave out the path option, you will likely run into an error when invoking the endpoints tool. To simulate this error within the Scala backend, delete the path option from the list method of the GamePlay controller, compile the code and invoke the endpoints tool on the GamePlay controller. Another fact that is not entirely clear from the documentation regards the *@Named* annotation. Essentially, a parameter that is "named" is one that appears in the URL in a form like *http://www.endpointapp.com/find?parameter1=blah&parameter2=blah*. A parameter that isn't named is simply included with the POST request data.

The annotations are necessary for the endpoints tool to be able to automatically generate the API discovery and configuration files needed to host your API backend on App Engine. After annotating your API backend, you need to use the endpoints tool to generate these API files. You invoke the endpoints tool like so:

```
/path_to_gae_sdk/bin/endpoints.sh get-client-lib <class_names>
```

For the <class_names>, you need to pass in the full class name of each class that implements any of the APIs you want to expose. If you have multiple classes, you must separate each class with a space. The endpoints tool also automatically generates a Java client library that is ready to be used with Android and Java web apps. To generate a client library for an iOS application, another step is required that uses one of the API discovery files, which is discussed in the next section on *Objective-C and iOS*.

However, if you try to use the endpoints tool on Scala code, you will get all kinds of Java exceptions about missing classes. These exceptions are due to the fact that the CLASSPATH of the endpoints tool has none of the Scala JARs and thus none of the Scala specific classes are recognized at compile time. Luckily, there is a simple fix to this issue: you simply need to modify the *endpoints.sh* script to include the Scala JARs on the CLASSPATH. This modification can be achieved by adding the following lines to the end of the *endpoints.sh* script, replacing *path_to_scala* with the full path to the Scala directory on your machine:

```
1    SCALA_LIB="/path_to_scala/libexec"
2    for jar in "$SCALA_LIB"/lib/*.jar; do
3            CLASSPATH="$CLASSPATH:$jar"
4    done
```

Once you have generated all of the API files, move all files other than the *.zip* into your *war/WEB-INF* directory. You can now test your API backend locally using curl. Another way to test your API that isn't explicitly noted in the Java documentation is by using the APIs Explorer tool. The really nice part about this tool is that it runs right in your browser and essentially provides you with an interface to call your various APIs. You can access this tool on API backends running locally, as well as those that are live, simply by adding the following path to the end of your application's URL: */_ah/api/explorer*. For example, if you are running your API backend locally on port 8888, you would enter the following URL into your browser: *http://localhost:8888/_ah/api/explorer*. You can create and pass in

the parameters your APIs require and even authorize requests through OAuth 2.0 using this wonderful tool.

Generating Client IDs

In order for other applications to be able to access your APIs, they need a client ID. You must create and include this ID within both the client application and the API backend. There are two different types of client IDs: simple and authorized. As the names suggest, a simple client ID provides unauthenticated access, while an authorized client ID provides authenticated access. You should use simple client IDs when you don't need to access user data and authorized client IDs whenever you need to access private user data. To create client IDs, you need to have a project in the Google APIs console. If you don't have one, it is very simple and straightforward to create a new one. Once you open this project, click on the "API Access" tab to create and manage client IDs.

Objective-C and iOS

This documentation assumes that you are already familiar with using Objective-C to write iOS applications. Apple has wonderful documentation to help familiarize you with Objective-C and iOS and there is a huge developer community online as well. The TicTacToe client is an iOS application that accesses the APIs provided by the Scala App Engine endpoint using the client library generated by the endpoint tool. This client library relies heavily on the Google APIs Client Library for Objective-C. The Objective-C client library can be used for more than just accessing APIs exposed by a cloud endpoint, such as integrating an iOS application with Google Drive and Google Calendar. However, this documentation is only concerned with accessing cloud endpoints APIs from the iOS client.

Linking to the Google APIs Client Library for Objective-C

For the purpose of the TicTacToe iOS client, the Objective-C client library was solely used because of the dependency from the auto-generated iOS client library discussed in the next section. While there are a few different ways to use the library within an iOS application, by far the easiest approach is to simply compile the source files directly into your application. To check out a copy of the Objective-C client library, type the following command at the command line:

```
svn checkout http://google-api-objectivec-client.googlecode.com/svn/trunk/ google-api-
↪objectivec-client-read-only
```

Once you have run this command, you should have a directory with a name like *google-api-objectivec-client-read-only*, inside of which is two more directories; one with examples showing the different services being used and the other with the source code. All you have to do is drag all of the header and implementation source files that your application will need to make use of into the navigator pane of your iOS application in Xcode. If you take a look at the TicTacToe iOS client, you will see two folders within the "Supporting Files" folder, one for the headers that the application needs and one for the source. However, if your iOS application uses ARC (*it should!*), you need to modify the command line arguments used when the Google client library files are compiled because the client library does not use ARC. To do so, select the target you added the library files to and go to the "Build Phases" tab. From this tab, expand the "Compile Sources" section, select all of the library files and press enter. In the text area that pops up, type "-fno-objc-arc" and press enter once more. Now when you build your application, you should not receive any errors regarding these files. After you have successfully added the library to your application, all you need to do is *#include* the headers in your application code where the classes are used.

Generating the iOS Client Library

As eluded to earlier, generating the iOS client library from your annotated API backend requires a bit more work than with the Java client library. You still must invoke the endpoints tool on your annotated APIs, but you also need to make

use of a Service Generator tool that comes with the Google APIs Objective-C client library. The Service Generator Xcode project is located within the *Source/Tools/ServiceGenerator* directory and you need to use this Xcode project to build the tool itself. Once you open the *ServiceGenerator.xcodeproj* file, just build the project (hotkey *command+b*) and the Service Generator tool will be built and placed in the "Products" folder in the navigator pane of the Xcode project. Click on the *ServiceGenerator* executable in Xcode and, from the utilities view, use the file inspector to locate its full path (likely in the derived data for the project). You then use the tool like so, passing in the *xx-xx-rpc.discovery* file autogenerated by the endpoints tool:

```
/full/path/to/ServiceGenerator apiName-version-rpc.discovery --outputDir API
```

This command will generate the necessary Objective-C classes that provide basic consumption of the APIs you exposed. If you wish to add custom methods of your own to these classes, you shouldn't write this code directly into the auto-generated files. Instead, you should subclass the auto-generated classes and add the new functionality in the subclass. The main reason behind this idea is that every time you change your API backend and recreate your client libraries, the new auto-generated files won't have the custom code you added, should any of those files change.

> **Note:** You need to explicitly create getter and setter methods for all of the instance variables that you want to be created as properties within the auto-generated code.

Objectify

Last, but certainly not least, is the Objectify framework. This framework is specifically designed to be used as a wrapper around the Java implementation of the Google App Engine datastore. With Objectify, you can easily configure your entities using annotations and interact with the App Engine datastore using a clean, simple API. You also have access to an Objectify session cache on top of the App Engine memcache and schema changes are made much easier using some tricks with Objectify annotations. All of this and more is explained in the documentation for Objectify for those who are unfamiliar. The TicTacToe application uses Objectify 4.0b3, which can be found here. You will also need to download the Google Guava JAR, as this is a dependency of this version of Objectify.

Using Objectify with Scala

One of the greatest features of Objectify 4 is its ability to work well with Scala. In fact, their integration "just works" excluding a few minor points of interest:

- To autopopulate the id field of a Long id, you need to directly use the java.lang.Long type because Scala's Long doesn't allow initalizing to a *null* value.
- The *type* keyword is a reserved word in Scala. To use the *type()* method of the Objectify service, you need to surround it with backticks: *'type'()*.
- The *type()* method expects a class type as input, which is retrieved using the *classOf[]* operator in Scala.

Using Objectify with Google Cloud Endpoints

A really nice feature of Objectify is the ability to define complex relationships between entities using the Key<?> class and the Ref<?> class. Unfortunately, the endpoints tool will throw an exception if any of the classes used as a parameter or return type in an API contain a Key<?> or Ref<?> member variable. The endpoints tool fails because it cannot properly serialize an instance of either of these two classes when the data is passing between the API backend and client. Although this limitation didn't cause any problems with the TicTacToe application, it is nonetheless a significant limitation. A suggested workaround for this issue would be to separate your entity classes into an internal representation that gets saved into the datastore and an external representation that is only used to pass data between the API backend and the client. The external representation would only need to contain the ID of the

internal representation, so that you could retrieve the internal representation in the API backend when a request comes in.

CHAPTER 3

Playing TicTacToe

The TicTacToe game is implemented such that a player can create a *PendingGame* and immediately play their first turn without having to wait for an opponent. When an opponent joins the game, the game becomes a *CurrentGame* and the *date* updates after every turn. When a game finishes, it becomes a *Score* and maintains an *outcome* that reflects the final game board.

The main view of the iOS client app is a table view that displays all of the users current and completed games once they sign in. The application is designed to be as scalable as possible. For example, selecting a current game will transition that game to the main screen and query the backend for the most recent copy. If the copy currently stored in the main table view of the client application is different from the one retrieved from the backend, the old copy is replaced with the updated copy. By doing so, the client application only needs to perform the more expensive list queries once after the user signs in. Likewise, while the client could poll the API backend to check for the other player's turn, it instead provides a refresh button that the user can press to query the backend for any updates to the game. Should it detect an update, the same process is applied to update the main table view as before.

All of the logic for checking the status of a game is placed client-side to further reduce the burden on the backend. When a game finishes and becomes a *Score*, it keeps the same ID, which ensures that a user won't see an error when a game they are currently viewing ends. The *outcome* property of every *Score* is a *String* structured so that unless the outcome is a tie, the first word is the email of the winner followed by "defeated" followed by the email of the loser. This structure allows the client application to display some more meaningful information to the users when viewing *Scores*.

CHAPTER 4

The Servlet Context Listener

A common need of many applications is the ability to execute some operations when the application is initialized. With Google App Engine, this need is satisfied through three different approaches: a <load-on-startup> servlet, a ServletContextListener, or a custom warmup servlet. Google's documentation on warmup requests already covers the basics of these three approaches.

TicTacToe needs to use a servlet context listener for two reasons: to register the datastore entities with Objectify and to initialize the application's global logger. The first reason is simply a requirement by Objectify and because it should only run once before all other servlet code, it fits nicely into the servlet context listener. The second reason is to ensure that the application's logs correctly reflect all of the messages logged within the application's code.

One important fact to remember about a ServletContextListener is that it might be invoked as part of the first user request to your application if no warmup request was initiated, which will result in loading requests. If you have some complex logic being handled within a ServletContextListener, your application will incur additional load time, which might not make your users too happy.

CHAPTER 5

Modeling The Entities

As mentioned in the Technologies section, the Objectify framework was used to interact with the App Engine datastore. When starting a project that uses Objectify, you first need to model the entities that your application will be placing into the datastore. You then annotate your entities properly, following the official documentation, and you are ready to start storing and loading your data.

Objectify Configurations

All of the entities used by the TicTacToe application inherit from a common base class named *Game* and can be found in the *Game.scala* file. The *Game* base class is labeled with the *@Entity* annotation, while the subclasses are labeled with the *@EntitySubclass* annotation; displaying Objectify polymorphism in action. The *index=true* specification on the annotations of the subclasses lets Objectify know that we wan't to query on these classes. Behind the scenes, Objectify is actually creating another property on our entities specifying their underlying class type. Objectify also requires all entities to have a no-arg constructor, which explains the private *this()* methods in all of the subclasses.

> **Note:** Scala creates a no-arg constructor for the Game base class implicitly.

Another important Objectify-related configuration lies in the use of the *@Index* annotations. As per the best practices, you should use indexes sparingly and only on properties that you will need to query on. For this reason, only the *user1*, *user2*, and *date* properties are indexed in TicTacToe's entities.

Limitations

One of the limitations of using Objectify is the lack of support for an abstract base class. The *Game* class should really be abstract and never directly instantiated, but adding the *abstract* keyword to the class definition causes a runtime exception to be thrown by Objectify. The exception essentially states that Objectify was unable to instantiate an abstract class without more specific type information. One possible workaround for this issue would be to simply remove the *@Entity* annotation from the abstract base class and place this annotation on each of the concrete subclasses. The disadvantages to this approach are that this will create a separate datastore kind for each of the subclasses and it requires an amount of code-repeat proportional to the amount of code in the abstract base class (since you have to override the member properties for Objectify to pick up on them).

While this workaround could have been implemented for TicTacToe, another limitation surrounding abstract base classes is introduced by using the Google Cloud Endpoints service. As evidenced by the *findOrCreateGame()* method of the *MatchmakingController* class, an abstract base class is useful because it can be used to return an object with a type corresponding to any of the concrete subclasses. However, because of how the endpoints tool generates the API client libraries and configuration files, the underlying type of the object returned by the *findOrCreateGame()* method will always be *Game*. When the client sends a *Game* object back to the API backend, it will have type *Game* and not one of the subclass types. If *Game* is abstract, this will cause the JSON serializer to throw an exception complaining about how it cannot serialize an abstract class.

Unfortunately, due to the limitations just mentioned, some nice Scala features like pattern matching cannot be utilized because the underlying type of a parameter with type *Game* will always be *Game*. While this doesn't severely affect the TicTacToe application, for a more complex application this lack of support might not be acceptable. Luckily, there will be support in the future for abstract base classes in Objectify, likely using an annotation such as *@EntityBase*. Once this support is released, the workaround mentioned in the Objectify section of the Techonologies page about having a separate internal and external representation could be used to workaround the issue introduced by Google Cloud Endpoints.

CHAPTER 6

The Backend API Implementation

The development process, as discussed in the Google Cloud Endpoints section of the Techonologies page, for an App Engine API backend is very straightforward. The TicTacToe application splits the API between three different controllers to manage the three unique aspects of the application: a *MatchmakingController* for the matchmaking system to find a game, a *GamePlayController* for managing game play and a *ScoreController* for displaying completed games. The APIs that these controllers implement all require authentication, evidenced by the *checkForUser()* method called at the start of each API.

Limitations

Certain limitations were introduced into the API backend due to App Engine restrictions and conflicting technologies. To allow App Engine to efficiently load values from the datastore, only queries with a certain structure are allowed to be executed. Google explicitly documents the restrictions on datastore queries and these restrictions introduced a few limitations within the TicTacToe application. Within the *findOrCreateGame()* method of the *MatchmakingController* class, the first limitation is introduced with the following line:

```
val pendingGames:MBuff[PendingGame] = ofy().load().`type`(classOf[PendingGame]).order(
↪"date").list().asInstanceOf[JList[PendingGame]].asScala.filter{ pg:PendingGame =>␣
↪pg.getUser1().getEmail() != user.getEmail() }
```

This code line is retrieving a list of all *PendingGame* objects that were not created by this user, i.e. all pending games that this user can join. This list is ordered by the *date* property of each *PendingGame* object such that the oldest pending game is first in the list. In this way, the player who has been waiting for a second competitor the longest will be the one whose game the new player joins. However, one of the query restrictions is that properties used in inequality filters (i.e. any filter besides ==) must be sorted first. This restriction would mean that we would have to order first by *user1* and then by *date* and players with emails that start with the character a would get the greatest priority. Luckily, it is safe to assume that this player won't have many *PendingGame* objects in the datastore and thus not much of an extra hit is taken simply by letting the datastore order by *date* and then filtering the result.

The other hoop to jump around is due to the inability of the endpoints tool to serialize Scala classes. This inability causes the endpoints tool to throw an exception when it runs into a method whose return type involves a class specific to Scala. Luckily, the fix for this issue is simple: the *scala.collection.JavaConverters* package provides a simple way

to convert back and forth between certain Scala and Java classes. Using this package, you can simply use *asScala* or *asJava* to accurately convert between certain supported types.

CHAPTER 7

The iOS Client

The TicTacToe iOS client has a simple Master-Detail interface with two different kinds of detail views. The master view is a table view with two sections representing the two different types of detail views: a current games section and a completed games section. As already mentioned, the iOS client relies heavily on the Google APIs client library for Objective-C and the code generated by the endpoints tool to interact with the API backend. One important difference that you may have noticed is that the API backend has three subclasses of the *Game* base class, while the iOS client only has two classes: a *GTLTicTacToeGame* class and a *GTLTicTacToeScore* class. This difference is due to a limitation already discussed on the Modeling the Entities page: many of the API methods take a parameter of type *Game* and return a value of type *Game*. The endpoints tool doesn't realize this is a base class and believes it to be the actual type to use for the object sent to and from the client. The *GTLTicTacToeScore* class gets created because of the *ScoreController* that directly deals with *Score* objects. Because the *PendingGame* and *CurrentGame* classes dont add any extra members, this behavior is acceptable for the TicTacToe application.

Even if the *PendingGame* and *CurrentGame* classes did add extra members, they would still be retrievable client-side through a method rather than a property. All of the properties on the object returned by the API backend are serialized into JSON before being passed on to the client. Using the *JSONValueForKey:* method defined in the *GTLObject.h* header, you can query for the value of a property by a string representing the name specified within your API backend (i.e. a string "date" could be used to retrieve the *date* property of a *GTLTicTacToeGame* object). This method can be seen in action in multiple methods in the *TTTScoreViewController* and in the *tableView:cellForRowAtIndexPath:* method of the *TTTGameListTableViewController* with a call to *JSONValueForKey:@"outcome"*. This method is used because the API backend method *getCurrentGameBoard()* always returns a *Game* object to the client, even if the underlying type is *Score*. Thus, when a *Score* object is retrieved from the backend through this method, it will be instantiated as a *GTLTicTacToeGame* object and won't respond to the *outcome* property unless you directly access the JSON array.

Asynchronous Nature of API Queries

One important thing to know about the API queries that the generated code executes is that they are all asynchronous. The query is fired off and the method immediately returns. When the API backend processes the request, the completion handler block is called and the result is returned to the client application asynchronously. You need to be careful to not let a user fire off multiple concurrent queries as they will be hogging the API backend and running up your costs to host your API backend on App Engine. The iOS client application handles this by simply disabling the user

interface while queries are executing and displaying an activity indicator to the user to indicate that some activity is occurring.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search