CHAPTER 14

Google Cloud Endpoints

The production and sale of electronic devices connected to the Internet has exploded in recent years. In 2014, enough mobile devices were sold to connect one sixth of the world's population. And this is just the beginning. Big players in the market are working on alternatives to meet the needs of communities and countries where, due to socioeconomic factors, the penetration of mobile devices is still low. Most importantly, it is not the number but the diversity of devices we are seeing that demonstrates the need to connect them all in an organized way. Intelligent houses, cars, home automation systems, sensor boards, and the rest of the spectrum covered by the term *Internet of things* are generating huge amounts of information about our world. But who is going to orchestrate all that?

This may be an ambitious and unnecessary problem for most system architects, developers, and other engineers to solve, at least this year. Nevertheless, when you design a system nowadays, it is relevant—and necessary in some cases—to account for transparently connecting different devices: mainly computers, but also mobile devices. This generally means if you develop an API to connect and sync all your clients, you need to enable each of them to communicate with your API in the cloud. This can become an arduous amount of work if you develop native applications. Suppose you add one endpoint to your API, you then need to write logic to communicate with this endpoint from each of your clients.

This is the point at which this chapter becomes useful to you. Google Cloud Endpoints is a service built on top of Google App Engine that allows you to reduce the overhead of creating, maintaining, and connecting to a RESTful API from your clients. One of its main features is the ability to decorate your API with the necessary information to extract access endpoints, request methods, the version of your API, request parameters and allowed client IDs, and so on. This information is then used to describe your RESTful API and automatically generate client libraries for Android, iOS, and JavaScript clients that you can import onto each platforms for which you develop. As mentioned, Cloud Endpoints operates on top of App Engine, and it is available for Java and Python; therefore, you can access all the APIs, services, and features that App Engine provides, some which are discussed in Chapter 5.

Cloud Endpoints and ProtoRPC

In Python, Cloud Endpoints uses the protocol RPC library. This library is one of the frameworks you can use to build your application; others are webapp2 and Flask. This framework allows you to implement HTTP-based remote procedure call (RPC) services, each made up of a collection of message types and remote methods to interact with web applications. Details about this library are not included in the chapter because most of it is abstracted away by the Cloud Endpoints library.

As you learned in Chapter 5, App Engine uses NDB in Python to control your data storage system or database. Recently, in order to integrate Cloud Endpoints without needing to deal with ProtoRPC messages to exchange information, a new library called `endpoints-proto-datastore`[1] was introduced. This library is used to simplify the logic of applications and improve readability.

In this chapter, you build an API similar to the one in Chapter 5. If you have not read that chapter yet, don't worry; you see every part of the application involved in the process.

Setting Up Your Environment: The SDK

■ **Note** You can skip this section if your system is ready to start working on a new application.

The Python SDK includes all the tools you need to build, test, and manage your application code, data, and indexes. You do that locally first, by using the development server included in the SDK. You operate the local server from the Google App Engine Launcher, a small UI-based application that helps organize App Engine projects, run them locally, access logs, explore their status through a browser-based console that resembles a small part of what you can expect from the online dashboard, and finally deploy them to the world. Alternatively, you can use the `dev_appserver.py` and `appcfg.py` commands to perform these and other tasks. We mention them throughout the book when you need their specific functionality. Also note that you can use these commands to integrate and automate specific operations in of your own development/deployment process.

You can find the Python SDK at `https://cloud.google.com/appengine/downloads`. It comes in the form of an installer file for Mac and Windows, and a zip file for Linux. Installing the SDK on Mac and Windows also gives you access to the Google App Engine Launcher UI.

As of this writing, App Engine only supports for Python 2.7. If your code is written in Python 2.5, which is now deprecated, consider migrating to 2.7. You can find a complete guide to how to do that at `http://cloud.google.com/appengine/docs/python/python25/migrate27`.

To make sure Python is installed on your computer, simply run the following command on your console or terminal:

```
python -V
```

If the output looks something like `Python 2.7.x`, you are good to go. Otherwise, you need to go to the Python web site to download and install Python 2.7 on your machine: `https://www.python.org/downloads/releases/python-279`.

The Foundations of Your Application: app.yaml

As a quick reminder, the application you are building in this chapter is a replica of the API for the LunchMates application, which you built on App Engine in Chapter 5. LunchMates is a service that helps individuals who share a common interest in a given topic to get together to learn and share knowledge about that topic. Users are expected to create informal meetings such as lunch, drinks, brunch, and so on, and define a place and time for their events; potential lunch mates can check the map of their area to look for meetings. They should be able to send requests to join meetings and be accepted or rejected as quickly as possible.

[1]You can find `endpoints-proto-datastore` in this repository on GitHub: `https://github.com/GoogleCloudPlatform/endpoints-proto-datastore`.

The app.yaml file is the entry point for each request. In it, you specify the version, application identifier, libraries, and request handlers, but it is also the place to configure many other application features. For details, see Chapter 5.

For this API, the app.yaml is as follows:

```
version: 1
runtime: python27
threadsafe: yes
api_version: 1

handlers:

# Endpoints handler
- url: /_ah/spi/.*
  script: main.app

# Endpoints libs
- name: pycrypto
  version: "2.6"
- name: endpoints
  version: 1.0
```

In the first group of parameters, the version helps differentiate iterations of the application. handlers are the agents that process requests. For Cloud Endpoints, the handler prepended with /_ah/spi/ is the one that manages requests. Note that the URL specified must not change, because this is a preconfigured value.

Finally, you are importing two of the libraries included in the SDK. pycrypto is used to deal with signatures and tokens, and endpoints contains the necessary logic to take advantage of this service.

Your API and api_server

According to the app.yaml file you just defined, the script that handles Cloud Endpoints requests is called main.app. This looks for the variable app in the main module (main.py):

```
import endpoints

# Controllers
from controllers import base

from controllers import meetings
from controllers import meeting_requests
from controllers import users
from controllers import auth

# Endpoints API
app = endpoints.api_server([base.lunchmates_api], restricted=False)
```

Cloud Endpoints is expecting an instance of endpoints.api_server. The classes added to the first argument of this method determine the submodules that conform to your API. These submodules are classes that inherit from remote.Service and implement endpoints.api directly or are decorated as api_classes of a parent endpoints.api. Set the restricted keyword to False so that external clients can access your API.

■ **Note** If a single class defines your API, it is recommended that you put the code for your API class and API server in the same file. Not only is your code easier to read, but both objects share common library symbols and imports. For example:

```
# imports
....
@endpoints.api(name='lunchmates', version='v1', description='LunchMates API')
class LunchMatesApi(remote.Service):
    # methods
....
app = endpoints.api_server([LunchMatesApi], restricted=False)
```

Going back to the API you are building, the `endpoints.api` object is referenced as an instance variable (`base.lunchmates_api`) instead of a class (`LunchMatesApi`). This is because this instance is simply a placeholder variable that decorates each of the API classes that conform your multiclass API. In this case, you want all the controllers—`meetings.Meetings`, `meeting_requests.MeetingRequests`, `users.Users`, and `auth.Auth`—to be part of the same API. Therefore, they are all decorated with `@lunchmates_api.api_class(....)`. This API variable is defined in a file called `base.py` in `controllers/`, along with a couple of utility methods used by the rest of the endpoint controllers:

```
#!/usr/bin/env python

import endpoints
from model.model import UserData

# Client IDs
WEBAPP_CLIENT_ID = 'your-webapp-client-id.apps.googleusercontent.com'
IOS_CLIENT_ID = 'your-ios-client-id.apps.googleusercontent.com'
ANDROID_CLIENT_ID = 'your-android-client-id.apps.googleusercontent.com'
ANDROID_AUDIENCE = WEBAPP_CLIENT_ID

lunchmates_api = endpoints.api(name='lunchmates', version='v1',
                              description='LunchMates API',
                               allowed_client_ids=[
                                      WEBAPP_CLIENT_ID,
                                       IOS_CLIENT_ID,
                                       ANDROID_CLIENT_ID,
                                      endpoints.API_EXPLORER_CLIENT_ID],
                               audiences=[ANDROID_AUDIENCE])

def authenticated_user_data():

    current_user_data = UserData.query(
        UserData.auth_user == endpoints.get_current_user()).get()

    if current_user_data is None:
        raise endpoints.UnauthorizedException()

    return current_user_data
```

lunchmates_api defines the main aspects of your API. endpoints.api accepts the arguments listed in Table 14-1.

Table 14-1. *endpoints.api Arguments*

Argument	Description
name	Used to construct the path you need to query to access your API. The name must start with a lowercase letter and match the following regular expression: [a-z]+[A-Za-z0-9]*. All services in your API share the same path. In the previous example, this would be <app-id>.appspot.com/_ah/api/lunchmates/<version>/<service-method-path>
version	Determines the version of your API.
description	A short description of your API, which is used, for example, in the API's discovery service. This service allows you to make requests and test your API.
allowed_client_ids	Specifies which clients are allowed to access information exposed through authentication. Only these are granted an access token when requested. You can create identifiers for your clients in the Developers Console by selecting the project or creating a new one, navigating to APIs & Auth ➤ Credentials, and clicking Create New Client ID. To read more about OAuth2 and client IDs, see Chapter 3.
audiences	Helps you avoid token requests to access your API coming from unwanted sources in your Android device. The audience for your Android client is the client ID for your web application.
canonical_name	Used to name API classes in the automatically generated client libraries.
documentation	A string containing the URL where users of the libraries can find documentation about them.
owner_domain	The domain name representing the entity that owns the API: for example, lunchmates.com. The full package path is generated from the combination of this field and package_path, in reverse order. That is com.lunchmates for this example.
owner_name	The name of the API's owner.
package_path	A set of string values separated by a forward slash (/) to further specify your package. Used in conjunction with the owner domain, it determines the full package name used in your client libraries. For example, if your owner_domain is set to lunchmates.com and the package_path is api/tasks, the resulting package name is com.lunchmates.api.tasks.
scopes	The list of scopes to be used when authorizing access from Cloud Endpoints. If you do not specify anything, the default value is https://www.googleapis.com/auth/userinfo.email. To read more about OAuth, scopes, and the authorization process, check Chapter 3.
hostname	Specifies the hostname of your App Engine application. The default value is <application-id>.appspot.com. For this example, that is lunch--mates--endpoints.appspot.com.
title	The text used to display a title for your application in services like APIs Explorer (<app-id>.appspot.com/_ah/api/explorer) or discovery services.

The method `authenticated_user_data` is useful when a specific part of your application needs to retrieve the associated metadata for a given user.

The model: A Bridge between NDB Datastore and Cloud Endpoints

Python has always been one of the programming languages leading the development process for App Engine. Because of that, it receives services and features before Java, Go, or PHP, and its libraries are in a more advanced development state. NDB Datastore is a good example. This is the layer that allows you to communicate with Google Cloud Datastore, a schema-less, noSQL data-storage system build on top of Google Cloud BigTable to perform and scale. NDB Datastore includes features like internal memory caching to speed up requests, a wide set of properties for models that helps you optimize how you store and retrieve data, support for transactions and asynchronous operations, and so on.

On the other hand, Cloud Endpoints makes it very easy to generate API code and take advantage of automatic client-library generation to reduce the friction and investment of targeting multiple native platforms at once. However, as mentioned earlier, Cloud Endpoints is built on top of ProtoRPC services, leaving an important gap in terms of integration and migration for applications using any of the other common frameworks used in App Engine—webapp2, Django, Flask, and so on—and/or relying on NDB to build the model.

This is where `endpoints-proto-datastore`[2] becomes really useful. This library provides you with tools to develop your application without having to think too much about ProtoRPC messages. Instead, you can use your own NDB models to operate your services directly. As of this writing, this library is not included in Cloud Endpoints. Therefore, to use it, you need to add it to your application.

■ **Pro Tip** If you do not want to contaminate your code repository, you can add `endpoints-proto-datastore` as a submodule or dependency so that the source control and life cycle of your code and external libraries remain independent. For example, in `git`, you can add a submodule running the command `git submodule add https://github.com/GoogleCloudPlatform/endpoints-proto-datastore` from the folder where you intend to store your libraries—for example, `libs/`.

Before taking advantage of the potential of `endpoints-proto-datastore`, there is only one thing you need to do. Your model objects need to inherit from `endpoints_proto_datastore.ndb.EndpointsModel`. That simply means replacing `ndb.Model` with `EndpointsModel`, because the latter inherits from `ndb.Model`.

Whether due to further optimizations or type inconsistencies between ProtoRPC messages and NDB Datastore, you also need to migrate a few properties to `endpoints-proto-datastore`, as listed in Table 14-2.

[2]You can find the Endpoints Proto Datastore in this repository on GitHub: `https://github.com/GoogleCloudPlatform/endpoints-proto-datastore`.

Table 14-2. *Properties to migrate from `ndb.Model` to `endpoits-proto-datastore`*

Property in NDB	Property in Proto Datastore	Reasoning
`DateTimeProperty`, `DateProperty`, and `TimeProperty`	`EndpointsDateTimeProperty`, `EndpointsDateProperty`, and `EndpointsTimeProperty`	New properties in Proto Datastore that allow a keyword argument with the `string_format` of the date, to handle serialization internally.
`ComputedProperty`	`EndpointsComputedProperty`	Used to explicitly set the type of the value generated in this property, because it cannot be extracted from the implementation of `ndb.ComputedProperty`. If no `property_type` argument is specified, the default is used: `messages.StringField`.
`UserProperty`	`EndpointsUserProperty`	The final entity remains unchanged. `EndpointsUserProperty` takes care of retrieving the authenticated user using the `endpoints` library: `endpoints.get_current_user()`.
`IntegerProperty` and `FloatProperty`	`EndpointsVariantIntegerProperty` and `EndpointsVariantFloatProperty`	Allow custom serialization of integers and floats, respectively, by accepting variant types if they were used to create a message field.

This is what the model used in the API for the `LunchMates` application built in Chapter 5 looks like now:

```
#!/usr/bin/env python

import unicodedata

from protorpc import messages

from endpoints_proto_datastore.ndb import EndpointsModel
from endpoints_proto_datastore.ndb import EndpointsAliasProperty
from endpoints_proto_datastore.ndb import EndpointsDateTimeProperty
from endpoints_proto_datastore.ndb import EndpointsComputedProperty

from google.appengine.ext import ndb

DATE_FORMAT_STR = '%Y-%m-%dT%H:%MZ'
```

```
class BaseModel(EndpointsModel):
    created = ndb.DateTimeProperty(auto_now_add=True)

class UserData(BaseModel):

    def normalize(self):
        return unicodedata.normalize(
            'NFKD', unicode(self.name)).encode('ascii', 'ignore').lower()

    _message_fields_schema = ('created', 'id', 'auth_provider', 'name', 'email')

    auth_provider = ndb.StringProperty(
        choices=['google', 'facebook'], required=True)

    name = ndb.StringProperty(default='')
    search_name = EndpointsComputedProperty(normalize)
    email = ndb.StringProperty(required=True)
    auth_user = ndb.UserProperty()

class Meeting(BaseModel):
    owner = ndb.KeyProperty(kind=UserData, required=True)
    venue_forsquare_id = ndb.StringProperty(required=True)
    location = ndb.GeoPtProperty()
    earliest_possible_start = EndpointsDateTimeProperty(
        required=True, string_format=DATE_FORMAT_STR)
    latest_possible_start = EndpointsDateTimeProperty(
        string_format=DATE_FORMAT_STR)
    topic = ndb.StringProperty(required=True)
    type = ndb.StringProperty(required=True, choices=['drink',
                                                      'lunch',
                                                      'brunch'])
    tags = ndb.StringProperty(repeated=True)

    @EndpointsAliasProperty(property_type=messages.IntegerField)
    def owner_id(self):
        return self.owner.id()

class MeetingRequest(BaseModel):

    meeting = ndb.KeyProperty(kind=Meeting, required=True)
    state = ndb.StringProperty(default='pending', choices=['pending',
                                                           'accepted',
                                                           'rejected'])
```

As you can see, most of the logic remains the same. The model now inherits from `EndpointsModel` instead of `ndb.Model`. `EndpointsDateTimeProperty`-ies include a default date format that is used to serialize without your needing to care about it, and `search_name` is now an `EndpointsComputedProperty`.

■ **Pro Tip** Override the value for `_message_fields_schema` in `EndpointsModel` to set a custom message schema used to serialize your model objects into ProtoRPC messages. You can assign a dictionary, a list, a tuple, or an instance of `MessageFieldsSchema` to it. If it is set to `None`, all properties and alias properties are included in the message.

If you take a closer look at the `Meeting` class, you see a method with a decorator that you have not seen before. `EndpointsAliasProperty` can be used to decorate methods that represent properties of your model that you do not want to persist in Cloud Datastore. This property can look similar to `ComputedProperty` in the sense that it allows you to serialize and access volatile information, potentially constructed from other properties in the model. In this case, there is a bit more to it. `EndpointAliasProperty` exposes a keyword to define a `setter` that is called every time a new value is assigned to the property. You can use this as a pre-hook to do some work before a new value is assigned. Notice that the same applies at read time: every time the property is accessed, the method you decorated with `EndpointAliasProperty` is called. Here is an example, extracted from the final model in the `LunchMates` Cloud Endpoints API:

```
class MeetingRequest(BaseModel):
....

    def ParentMeetingSet(self, value):
        meeting_key = ndb.Key(Meeting, int(value))

        # Assign key to meeting. Scenario A: new entity
        self.meeting = meeting_key

        # Add the key to query info. Scenario B: fetch entities for a given meeting
        self._endpoints_query_info.meeting = meeting_key

    @EndpointsAliasProperty(required=True, setter=ParentMeetingSet,
                            property_type=messages.IntegerField)
    def meeting_id(self):
        return self.meeting.id()
```

The method `meeting_id` has been decorated with `EndpointsAliasProperty`. This means every time this property is accessed—`meeting_request.meeting_id`—the value read is the result returned by the method in question. In this case, it returns the identifier from the key of the associated meeting. Also notice that a `setter` keyword has been provided, pointing to a method defined just before. This method is called when the property is set, which can happen either internally or from your own code. When the meeting ID for a request is set, this method creates a key with `kind=Meeting` and the identifier provided and assigns it to the meeting's `ndb.KeyProperty` in the model. In addition, the same key is added to the model's query information.

This can be confusing. Why is all that happening? The answer is simple. We'll explain it from the perspective of each of the use cases. There are two situations where you may want to set the `meeting` property in your model:

- When you create a new meeting request for a given meeting: In this case you need to create a `Key` for the given `meeting_id` and assign it to the new entity's `meeting` property.
- When you are querying for meeting requests for a given meeting ID. In this case, you want to make sure you specify the meeting that you want to use to filter your query.

Note that although this approach can be confusing at first, it holds a lot of potential for the way you develop your application. In the next section, you discover why `EndpointsAliasProperty`-ies are so powerful in the context of Cloud Endpoints Proto Datastore.

■ **Note** As you have seen, `EndpointsAliasProperty`-ies can also be assigned inherited keywords such as `repeated`, `required`, `default`, and or `name`.

Services and Request Handlers

If you have read Chapter 5 of this book, you probably remember the concept of *request handlers* as classes that process and respond to requests depending on their path, method, and parameters. The same idea applies to ProtoRPC and Cloud Endpoints. In this case, the handlers are referred to as *services*. These services contain methods that are responsible for processing incoming requests and returning a result. In Cloud Endpoints, in single-class APIs, services are decorated with the `endpoints.api` method. When your API has multiple `remote.Service` classes to process requests, you decorate each of them with `<your_api>.api_class`. This method accepts parameters similar to those of the `endpoints.api` method—`resource_name`, `audiences`, `scopes` and `allowed_client_ids`—but in this case they are only applied to the scope of the defined class. In addition to those mentioned, you can also provide a `path` argument that is added to the URL path and prepended to each method of the service. If you do not specify it, it defaults to `None`, which causes it to be omitted in the path. Suppose, for example, that based on the configuration of your API, your base URL is

```
lunch--mates--endpoints.appspot.com/_ah/api/lunchmates/v1/.
```

Now you add two new services to your API. The first one contains its own path:

```
@lunchmates_api.api_class(path='meetings')
class Meetings(remote.Service):

    @endpoints.method(....)
    def list(self, query):
        ....

    @endpoints.method(....)
    def create(self, meeting):
        ....
```

These methods can be accessed with the following URLs, respectively:

```
lunch--mates--endpoints.appspot.com/_ah/api/lunchmates/v1/meetings/list
lunch--mates--endpoints.appspot.com/_ah/api/lunchmates/v1/meetings/create
```

Next, consider a service for which you do not specify a path:

```
@lunchmates_api.api_class()
class Auth(remote.Service):
    @endpoints.method(....)
    def authenticate(self, user):
        ....
```

Because a path is not specified in `api_class`, this method is accessed using its name:

```
lunch--mates--endpoints.appspot.com/_ah/api/lunchmates/v1/authenticate
```

■ **Note** `endpoints.method` accepts a `path` argument. As you can see in the previous example, if this argument is not specified, the name of the method is used to construct the URL path.

The previous examples show how you must decorate your methods for Cloud Endpoints to be able to generate client libraries that can access them. `endpoints.method` accepts the arguments listed in Table 14-3.

***Table 14-3.** Accepted arguments in the `endpoints.method` decorator*

Argument	**Description**
`Request Message Class`	The message class corresponding to the ProtoRPC request used in the method. This argument accepts the class itself or its name.
`Response Message Class`	The message class corresponding to the ProtoRPC response used in the method. This argument accepts the class itself or its name.
`path`	A string with the path that is appended to the URL to access this method. If you do not set this argument, the method name is used.
`name`	An alternative name for this method. The name must start with a lowercase letter and match the following regular expression: `[a-z]+[A-Za-z0-9]*`.
`http_method`	HTTP method of the incoming request. Defaults to `POST` if not set.

■ **Pro Tip** Avoid collisions between the `resource_name` of your API and the `name` of your methods. For example, when generating the client library for Java, the part of the method name before the dot (`.`) is an inner class of the API resource name. Therefore, an API class with `resource_name='meetings'` and a method with `name='meetings.create'` would create a conflict and not be able to build.

In addition to the arguments in the table, you can specify the keywords audiences and `allowed_client_ids` at the method level, in which case they override the values specified in the API. You do so through `endpoints.api()` or `endpoints.api_class()`.

However, because you are using the Endpoints Proto Datastore API on top of Cloud Endpoints, most of the time you will not decorate with `endpoints.method` directly. Instead, the Endpoints Proto Datastore API provides two methods that simplify your requests by handling queries and serialization of Cloud Datastore model objects into ProtoRPC messages for you. Because of that, you do not need to specify the classes for the request and response `Message` objects.

■ **Note** Both `EndpointsModel.method` and `EndpointsModel.query_method` act as wrappers around `endpoints.method`, which is called at the end of both implementations.

EndpointsModel.method

`EndpointsModel.method` takes the `EndpointsModel` class specified and uses it to serialize and deserialize the entity into RPC messages. In addition to the arguments for `endpoints.method` listed previously, you can also set the arguments in Table 14-4.

Table 14-4. *Accepted arguments in the `EndpointModels.method` decorator*

Argument	Description
`request_fields*`	List of fields used to define the ProtoRPC message for the request.
`response_fields*`	List of fields used to define the ProtoRPC message for the response.
`user_required`	Accepts a boolean value. When set to `True`, an authenticated user is required to proceed with the request. If no user is present, the request responds with `401 Not Authorized`.

**Both `request_fields` and `response_fields` are optional arguments that accept a list, a tuple, a dictionary, or an entity of the class `MessageFieldsSchema`. If not set, messages are built using the variable_message_ `fields_schema` in your `EndpointsModel` class or the properties in the model definition if the schema is not set.*

Here is an example of a `POST` method used to create new meetings in the application:

```
from protorpc import remote

from base import lunchmates_api
from base import authenticated_user_data

from model.model import Meeting

@lunchmates_api.api_class(resource_name='meeting')
class Meetings(remote.Service):

    @Meeting.method(path='meetings', name='meetings.create', user_required=True)
    def create(self, meeting):

        meeting.owner = authenticated_user_data().key
        meeting.put()
        return meeting
```

The `Meeting` class—which inherits from `EndpointsModel`—is used to decorate the method. As explained earlier, this defines the underlying ProtoRPC message used in the request and response based on the `Meeting` model. The rest of the arguments are common to `endpoints.method`. Notice that an authenticated user is required to execute the request. Also note that the `path` is set to `'meetings'`. Hence the URL to access that method is

```
POST [localhost | <your-application-id>.appspot.com]/_ah/api/lunchmates/v1/meetings
```

The `meeting` object that is passed as an argument of the method you define is the result of deserializing the body of your request into the class specified on its decorator, `Meeting`. All that is left is persisting the object. In the previous example, the owner of the meeting is set using the currently authenticated user before the entity is persisted.

EndpointsModel.query_method

In contrast to the previous method decorator, `query_method` is intended to help you work with Cloud Datastore queries. As a result, only `GET` requests are allowed. When using this decorator, a query is created based on the parameters passed and is made available to you through the `query` property on your decorated method definition. In this method, you can modify the query as you wish—for example, adding a default sort order. You must return the query at the end of the method. This causes the query to be executed. In the response body of your requests there are two fields:

- `items` is an array of results of the type of the `EndpointsModel` class used to decorate the method. The number of results is limited by default, but you can configure it through the method's keyword arguments.
- `nextPageToken` is used as a means of pagination. You can use this value, adding it as a query parameter to your method so that if the caller sets it, it is converted into a cursor and used to return the next page of results.

This is an example of a simple `GET` method to fetch meeting requests:

```
import endpoints
from protorpc import remote

from base import lunchmates_api
from model.model import MeetingRequest

@lunchmates_api.api_class(resource_name='meeting_request', path='meetings')
class MeetingRequests(remote.Service):

    @MeetingRequest.query_method(path='requests', user_required=True,
                                 name='meeting_requests.list')
    def list(self, query):
        return query.order(-MeetingRequest.created)
```

Notice that the only modification to the query is adding an `order` clause. In this case, the results are sorted by the `created` property in descending order. Because `created` is a date, new records are returned first.

In addition to the arguments included in endpoints.method, query_method accepts the keyword arguments listed in Table 14-5, all of them optional:

***Table 14-5.** Accepted arguments in the EndpointModels.query_method decorator*

Argument	Description
query_fields	Determines the fields used to construct the query. Each field specified is expected to affect the results of the query. Because of that, the fields have to refer to properties that are part of the model in question—@Model.query_method—whether they are regular properties or EndpointsAliasProperty-ies. For example, to retrieve meetings of the type lunch, adding the query field type allows requests of the kind request_url?type=lunch. This adds the filter to the query for you and returns the results you are looking for. Note that this behavior only applies to equality filters. If you need something more elaborate, you can use EndpointsAliasProperty-ies with a setter to define the details of your query. Check out the MeetingRequest model to see an example.
collection_fields	Determines the fields to return for each item in the response results. If not specified, it defaults to the model schema.
use_projection	A boolean value that, when set to True, queries for a projection of the entity instead of the entire entity. The projection is defined using the collection fields. Projection queries are faster to retrieve and transfer, but remember that they need an index on each of the fields specified. You learn more about projection queries in Chapter 9.
limit_default	Determines the number of records to be returned in the response body when no limit clause is specified on the query. Defaults to endpoints_proto_datastore.ndb.model.QUERY_LIMIT_DEFAULT, currently set to 10.
limit_max	Sets the maximum allowed number of records to be returned in the response body. If a query attempts to limit the number of results with a number greater than the value specified in this argument, an endpoints.ForbiddenException is raised. Defaults to endpoints_proto_datastore.ndb.model.QUERY_LIMIT_MAX, currently set to 100.
user_required	Accepts a boolean value. When set to True, an authenticated user is required to proceed with the request. If no user is present, the request responds with 401 Not Authorized.

EndpointsModel includes three convenient EndpointsAliasProperty-ies that you can use as query_fields in query_method like this:

```
@YourModel.query_method(query_fields=('limit', 'order', 'pageToken'), path='your_path')
def yourQueryMethod(self, query):
    return query
```

These alias properties process the three query fields so that, when added as query parameters in the request URL, they are used to modify the query that returns the results you are looking for. Therefore, if you access this method with the following URL, your query returns five results from the selected page ordered by the created property:

```
<method_url>/?limit=5&order=created&pageToken=<token-urlstring>
```

Use these predefined EndpointsAliasProperty methods[3] as an inspiration to construct your own based on your needs.

Here is an example of a query method with an EndpointsAliasProperty used in the LunchMates API.

In the endpoint that handles meeting requests, there is a method that fetches all requests for a specific meeting, passing the ID of the meeting as a parameter in the URL:

```
<api_host>/_ah/api/lunchmates/v1/meetings/<meeting_id>/requests
```

You can now access this property—meeting_id—by specifying it as a query_field in your method definition:

```
@MeetingRequest.query_method(query_fields=('meeting_id',),
                            path='{meeting_id}/requests',
                            user_required=True,
                            name='meeting_requests.list')
def list(self, query):
    return query.order(-MeetingRequest.created)
```

Now, in your model, you can define an EndpointsAliasProperty with a setter so that every time query_method is called, the property meeting_id is assigned. Because of that, the setter you define in the model is executed. Finally, in this setter, you can filter the potential query to fetch meeting requests, using the meeting ID provided, like this:

```
class MeetingRequest(BaseModel):

    ....

    def ParentMeetingSet(self, value):

        meeting_key = ndb.Key(Meeting, int(value))

        # Assign key to meeting. Scenario: new entity
        self.meeting = meeting_key

        # Add key to the query info. Scenario: fetch entities for a given meeting
        self._endpoints_query_info.meeting = meeting_key

    @EndpointsAliasProperty(required=True, setter=ParentMeetingSet,
                           property_type=messages.IntegerField)
    def meeting_id(self):
        return self.meeting.id()
```

[3]Line of the code in the endpoints-proto-datastore repository where the predefined query_fields aliases are defined: https://github.com/GoogleCloudPlatform/endpoints-proto-datastore/blob/master/endpoints_proto_datastore/ndb/model.py#L826.

EndpointsModel.method and EndpointsModel.query_method can be very powerful assets to make the methods in your services react wisely based on the type of requests your application receives and the data you are exposing. Experimenting with them as concepts of query_fields, EndpointsAliasProperty, and the internal wiring of Cloud Endpoints can make them a bit hard to learn and adopt at first. Here are some resources that can help you throughout the process:

- endpoints_proto_datastore code repository: https://github.com/GoogleCloudPlatform/endpoints-proto-datastore
- endpoints source code: http://code.metager.de/source/xref/google/appengine/python/lib/endpoints-1.0
- Introduction to the Endpoints Proto Datastore API: http://endpoints-proto-datastore.appspot.com

Finally, do not miss the documentation at Google Developers. This is a living entity that is regularly updated with the latest additions and pieces of advice.

The APIs Explorer

The APIs Explorer is a tool that allows you to browse, explore, and perform requests against most Google APIs in a visual way. It is very helpful when you are working with concrete APIs and need to experiment and see the details of each request you are interested in, such as the full URL path, the parameters allowed, the responses you can expect, and so on. You can access Google APIs Explorer at

https://developers.google.com/apis-explorer

It is even more useful in the context of developing your own API or application. To access the APIs Explorer for the application you are currently working on, simply navigate to the following URL when the development server is running:

```
http://localhost:<port>/_ah/api/explorer
```

You can also access the APIs Explorer for your deployed API by replacing the local host with its deployed counterpart:

```
http://<your-app-id>.appspot.com/_ah/api/explorer
```

For example, for the LunchMates API you have been working on throughout the chapter, it is as follows:

http://lunch--mates--endpoints.appspot.com/_ah/api/explorer

From there you can see all your services and API versions and test your methods through the endpoints generated based on the configuration of your API.

Generating Client Libraries for Your Application

One of the major benefits of Cloud Endpoints is the ability to generate client libraries that connect and make requests to your API. This is useful in the initial phase of client development and also in scenarios of continuous development and iterative approaches. When your API is still changing, Cloud Endpoints allows you to circumvent the need to manually change your client libraries according to API updates.

■ **Pro Tip** Because you are using the command line and common build tools to generate your libraries, you can easily automate this process with continuous integration tools like Jenkins and Travis. You can add hooks to your API code repository[4,5] so that every time the production branch is updated, the client libraries are automatically generated and made available to your client developers.

The command-line tool `endpointscfg.py` allows you to generate API discovery docs for your application as well as the client library you need to use in your client to connect to the API you just created. For example, to create the API discovery doc for application, do the following:

```
$ endpointscfg.py get_discovery_doc --format rpc
                controllers.auth.Auth controllers.users.Users
                controllers.meetings.Meetings controllers.meeting_requests.MeetingRequests
```

You use the same command to generate the client library for your Android application, by using `get_client_lib` instead of `get_discovery_doc`:

```
$ endpointscfg.py get_client_lib java -bs gradle
                controllers.auth.Auth controllers.users.Users
                controllers.meetings.Meetings controllers.meeting_requests.MeetingRequests
```

As you can see, the structure of the command is as follows:

```
endpointscfg.py <command> <language> <options> [services-list...]
```

Where:

- `command` can be either `get_discovery_doc` or `get_client_lib`, depending on whether you are generating the discovery docs or a client library for your application.
- `language` determines the target programming language in which you want your client library to be exported. This is only applicable to `get_client_lib`.
- `options` specifies the associated options that you can add to the two commands.
- `services-list` represents the list of services that conform your API.

[4]Git push hooks in Travis: `http://docs.travis-ci.com/user/getting-started/#Step-four%3A-Trigger-Your-First-Build-With-a-Git-Push`.
[5]Git push hooks in Jenkins: `https://wiki.jenkins-ci.org/display/JENKINS/Git+Plugin#GitPlugin-Pushnotificationfromrepository`.

These subcommands have a few options in common, as listed in Table 14-6.

Table 14-6. *Available options in the* `endpointscfg.py` *command-line tool*

Argument	Description
Common to Both Subcommands	
`-a` or `--application`	Specifies the path for the directory where `endpointscfg.py` looks to execute the desired command. By default, this is the current directory. Remember that this path must point to the root directory of your application—that is where your `app.yaml` file is placed.
`--hostname`	Determines the hostname of your application. The default is used if it is not specified. This is helpful, for example, when you need to test your application locally or you are using a different host than the default one: `<application-id>.appspot.com`.
`-o` or `--output`	Indicates the directory where the output files should be placed. By default, this is the current directory.
Specific to `get_client_lib`	
`-bs` or `--build_system`	Describes the type of bundle that is produced based on the build system you are working with. Currently, the options for Android are `gradle` and `maven`. If you omit this option, the default bundle is used. This bundle contains only the dependency libraries and a `source.jar`.
Specific to `get_discovery_doc`	
`-f` or `--format`	Selects the API protocol type used to export the discovery docs. Options are `rest` (the default) and `rpc`.

The process of using client libraries to access your API is different on each platform. For example, you can generate a Java client library directly using the `endpointscfg.py` command, whereas on iOS you need to generate your library from Xcode. Conversely, in JavaScript, you can dynamically load your API into the JavaScript-provided library. This chapter explains the Android case. For the JavaScript[6] and iOS[7] versions, refer to the documentation at Google Developers.

Accessing Your API from Your Android client

The first thing you need to do is export the client libraries in Java using the `get_client_lib` command:

```
$ endpointscfg.py get_client_lib java -bs gradle
                controllers.auth.Auth controllers.users.Users
                controllers.meetings.Meetings controllers.meeting_requests.MeetingRequests
```

[6]"Using Cloud Endpoints in a JavaScript Client," https://cloud.google.com/appengine/docs/python/endpoints/consume_js.

[7]"Using Cloud Endpoints in an iOS Client," https://cloud.google.com/appengine/docs/python/endpoints/consume_ios.

The bundle is generated, compressed, and saved in the output folder you specify or your current directory. Once generated, you can decompress this file and add it to the root folder of your Android application. In this case, a Gradle module was generated. In order for your Gradle build script in Android Studio to recognize this module, you need to include it in your `settings.gradle` file:

```
include ':app', ':lunchmates'
```

Here, `app` is the module of the application and `lunchmates` is the API for the `LunchMates` application you just generated. Once defined, you can go into the `build.gradle` script in your application module (`app`) and add it as a dependency:

```
....
dependencies {
    ....
    compile project(':lunchmates')

    compile 'com.google.api-client:google-api-client-android:1.20.0'
    compile 'com.google.android.gms:play-services:7.0.0'
    compile 'com.android.support:appcompat-v7:22.0.0'
}
```

Notice that you need to add two more dependencies to your project:

- Google Play Services libraries allow you access the Account Manager, which helps you handle the authentication process.
- The API client for Android includes a set of libraries that help you interact with Google and other APIs. When you use Cloud Endpoints, your API is built using similar conventions and technologies as APIs in Google, therefore this dependency also simplifies access to your API in tasks like networking, serialization, authorization, and so on.

In your manifest, make sure to add a permission to access the Internet. If you need to make authenticated requests, also include `get` accounts from the accounts service and generate authorization tokens using the Account Manager with `USE_CREDENTIALS`:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android" ....>

    <uses-permission android:name="android.permission.INTERNET"/>
    <uses-permission android:name="android.permission.GET_ACCOUNTS"/>
    <uses-permission android:name="android.permission.USE_CREDENTIALS"/>

    <application ....>
        ....
    </application>
</manifest>
```

To make requests on behalf of a user with an account set up in the Android device, you use the class GoogleAccountCredential, which deals with the entire process of obtaining an access token for the selected account:

```
private static final String ANDROID_AUDIENCE = "server:client_id:<your-web-client-id>";

private GoogleAccountCredential credential;
private SharedPreferences sharedPreferences;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(....);

    sharedPreferences = PreferenceManager.getDefaultSharedPreferences(this);
    credential = GoogleAccountCredential.usingAudience(this, ANDROID_AUDIENCE);

    checkSelectedAccount(sharedPreferences.getString("account_name", null));
}
```

To avoid users needing to select their preferred account every time they open your app, you store their selection in SharedPreferences,[8] Once the credentials and SharedPreferences members are instantiated, you can check whether a user already selected an account to authenticate your requests—in which case you try to fetch the current list of meetings—or prompt the user with the available accounts to start the authentication process:

```
private static final int ACCOUNT_PICKER_REQUEST_CODE = 2;

private void checkSelectedAccount(String accountName) {
    if (accountName == null) {
        startActivityForResult(credential
           .newChooseAccountIntent(), ACCOUNT_PICKER_REQUEST_CODE);
    } else {
        setupLunchmatesApi(credential, accountName);
        fetchMeetingsList();
    }
}
```

[8]SharedPreferences is a simple storage for keys and values on disk. This means of storage is very convenient for small, unstructured amounts of information that you need to persist across sessions. For more information, refer to the documentation at http://developer.android.com/reference/android/content/SharedPreferences.html.

If there is not an account already stored, a new `Activity` is started to show the user the available accounts in the Android device. When an option is selected, the `Activity` closes and calls back the original `Activity` through `onActivityResult`:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);

    switch (requestCode) {
    case ACCOUNT_PICKER_REQUEST_CODE:
        if (data != null && data.getExtras() != null) {
            String accountName = data.getExtras().getString(AccountManager.KEY_ACCOUNT_NAME);
            if (accountName != null) {
                checkSelectedAccount(accountName);
            }
        }
        break;
    }
}
```

If an account was selected, `checkSelectedAccount` is called, persisting the current account and creating the API object that you can use to make requests:

```
private void setupLunchmatesApi(GoogleAccountCredential credential, String accountName) {
    credential.setSelectedAccountName(accountName);
    storeAccountName(accountName);

    Lunchmates.Builder lunchmates = new Lunchmates.Builder(
            new NetHttpTransport(),
            new JacksonFactory(),
            credential);

    lunchmatesApi = lunchmates.build();
}
```

In this method, you set the selected account in the `credentials` object and construct your API object with it. You also persist the currently selected account in `SharedPreferences` with `storeAccountName (accountName)`.[9]

Finally, the method `fetchMeetingsList` retrieves a list of meetings from your API:

```
private void fetchMeetingsList() {
    ...
    try{
        MeetingCollection meetings = lunchmatesApi.meeting().meetings().list().execute();
```

[9]This and other methods have not been included in the book. To see the full source code of this Android application go to `https://github.com/GoogleCloudPlatformBook/lunchmates-android`

```
        final String result;
        if (meetings.get("items") != null) {
            result = meetings.getItems().size() + " meetings";
        } else {
            result = "0 meetings";
        }
        Log.i("# of meetings", result);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

This code should not run in the main thread, because it blocks execution until the request returns a response. Among the alternatives, consider using `AsyncTask`, `Loader`, or `ThreadPoolExecutor`. If you need requests to execute and wait until a response is given even if the application is sent to the background, consider using a service. Keep in mind that services help you to execute logic independently of the lifecycle of your views or `Activity`-ies. However, a service runs in the main thread; thus you still need to handle executing the operations in separate threads.

You can find, test, and contribute to the example application from the associated GitHub repository: `https://github.com/GoogleCloudPlatformBook/lunchmates-android`.

Summary

In this chapter, you have seen the power of App Engine combined with Cloud Endpoints, which together give you the ability to develop your API and client-side logic in much less time than with the usual approaches. The combination of these two services is powerful when you target more than one platform at the same time and also helps you make your API-side logic simpler, more maintainable, and flexible in the face of potential changes in your business logic.

Now you have the tools you need to build an entire application or service from scratch. It is your duty to experiment and try the list of projects, ideas, and prototypes you have had in mind for a while but never had the resources to address.