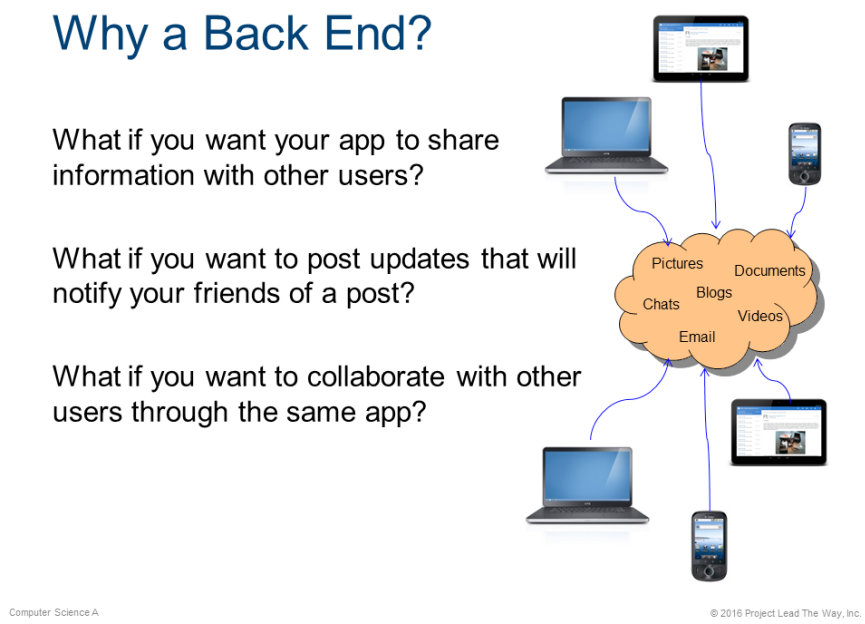
ACTIVITY **2.2.2**

Remote Database

INTRODUCTION

You have probably noticed that when you use an app and return later, your information from your last session is still there. This is called "data persistence" and it is a desirable quality. It is a nuisance for users to have to re-enter data every time they use your app.

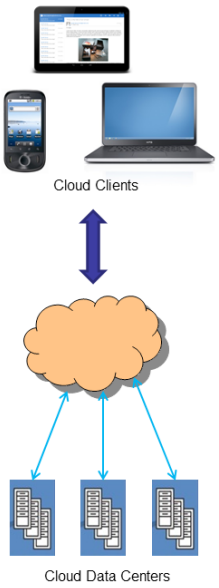


So far, you have created Android applications and stored any persistent data locally on your device. You can neither share it with other users nor access it from any other device. All the data, including text, images, and videos are stored on the device's internal storage.

To be able to share data and collaborate, you need to connect your app to a centralized database, which will hold all the data that your application users will create and can send notifications to users about news and updates. This database, and its supporting programs, provides a back-end service; it resides on a server that can be accessible to all app users over the Internet.

What Is BaaS?

- BaaS stands for Backend as a Service
- Also known as MBaaS: Mobile Backend as a Service
- BaaS data lives in physical servers
- Clients access BaaS from anywhere via the cloud
- BaaS frees development resources for other organizational needs



Computer Science A

Many software applications require a series of back-end services, like data storage, file storage, backup services, notification services, location services, search functionality, and other features that are commonly used across applications today. To save themselves the time and cost required to create and maintain these services, many organizations are using Backend as a Service. This service provides organizations with cloud-based services (i.e., services that live in the Internet) that meet all their back-end processing needs.

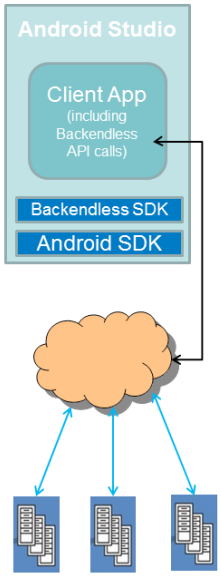
BaaS includes Software Developer Kits (SDK) and APIs that allow software developers to connect their front-end applications to back-end services, like cloud storage.

In spite of all the benefits listed above, there are still some concerns with modern computing, such as cost, having enough Internet bandwidth to support the organization's needs, and system/data security.

Connecting to the Web

Use Backendless for a back-end service

1. Install Backendless SDK for Android
2. Add Backendless SDK to your Android Studio™ project
3. Use Backendless APIs in your project to:
   - Connect to the back end
   - Query data
   - Create, update, and delete data



Computer Science A

Is This an Act of Magic?

How does code save the data about your objects to the cloud?

- Use Hypertext Transfer Protocol (HTTP)
  - HTTP Request: from client to server
  - HTTP Response: from server back to client
- HTTP request methods:
  - HTTP GET
  - HTTP POST
  - HTTP PUT
  - HTTP DELETE

Computer Science A

Data is sent over the World Wide Web using the Hypertext Transfer Protocol (HTTP).

All HTTP transactions occur through a request (sent from the client to the server) and a response (sent from the server back to the client requesting the data).

HTTP requests specify a method for the transaction. The most commonly used methods are:

HTTP GET: retrieve resources from the server

HTTP POST: create a new resource on the server

HTTP PUT: update existing resources on the server

HTTP DELETE: delete an existing resource on the server

Working with Android and Backendless

1. Design the App
2. Prototype the App
3. Build the front end (client app)
4. Integrate your client app with the back end using the Backendless APIs

Computer Science A

These are the general steps when working with Android and Backendless. You will learn these steps in more detail working through your Activities, Projects and Problems in this Unit.

There are a number of ways to achieve **data persistence** in your apps, but in this activity you will focus on one: storing data remotely in the cloud. To accomplish this, you will use a service called Backendless. Backendless is a remote **database** that provides you, as a developer, with much of the functionality that you need to store and retrieve data. For this reason, Backendless is referred to as a Backend as a Service (BaaS).

Data persistence
Information that is saved between multiple executions of a program.

database
A repository of data objects modeled using data classes. Backendless uses JSON format, so all data is stored as name-value pairs.

Sometimes apps require data sharing across multiple users and devices. The internal storage on a single device can be used to save the data locally, but for data to be shared between devices, there must be a centralized database on a network that all of the devices can access.

Consider all the data shared today on the World Wide Web—a good example is a social network. Multiple users post information that is made available instantly to their online communities. This is all accomplished using very large databases hosted on a server somewhere on the big network called the internet. Users from all over the internet can access this database and edit its data.

Though the use of the College App at this point in development would indicate using local storage, this course will use only remote data storage to achieve data persistence, to save time by teaching only one method.

Materials

- Computer with Android™ Studio
- Android™ tablet and USB cable, or device emulator
- Free Backendless account per student

RESOURCES

Activity 2.2.2 Visual Aid
Resources available online

Procedure

Part I: Storing and Retrieving Data

You will begin by modifying the `Profile` class within the College App for data persistence and then apply your learning to other classes within the College App.

abstract class
A class that cannot be instantiated and defines instance fields and methods for subclasses; differs from an interface mainly in that it can contain method implementations.

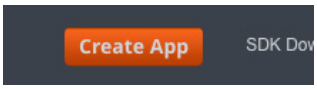
Interface
An abstract type that acts as a "contract" for classes, requiring them to provide implementation for stated methods.

1. To gain a base-level understanding of inheritance, **abstract classes**, and **interfaces**, work through the content at the following links, as directed by your instructor.
   a. **Object Oriented Programming Concepts**
   b. **Association vs. Inheritance**
   c. **Overriding an Inherited Method**
   d. **Using Super to Call an Overridden Method**
   e. **Access to Inherited Private Fields**
   f. **Inheritance and Constructors**
   g. **Abstract Classes**
   h. **Inheritance and Interfaces**
2. In Android Studio, open your College App from *Activity 2.2.1 Exceptions and Scope*. If you were unable to finish the activity, open *`2.2.1CollegeApp_Solution`* as directed by your teacher.
3. To demonstrate your understanding of interfaces and abstract classes, answer the following questions:
   a. A golf course has the following classes: Tree, Elm, Maple, Oak, Pine. Which, if any, are most likely abstract?
   b. A drawing program has the following files. For each, mark whether it most likely contains an interface, an abstract class, or a class:
      1. Pen.java
      2. Eraser.java
      3. Rectangle.java
      4. Tool.java
      5. PressureSensitive.java
      6. Fill.java
      7. VariableStrokeWidth.java
   c. You are designing an app that tracks the presence of pests and pest control in and around your house. Name as many interfaces, abstract classes, and classes as you can.

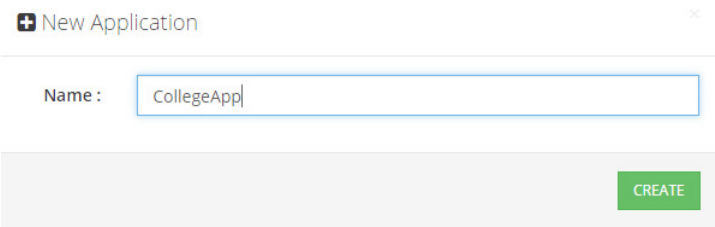
Interfaces	Abstract Classes	Classes

4. Log in to your Backendless account, or create one if you do not have one already at backendless.com. This is the service that you will use to host your data in the cloud.

5. To create a Backendless app, click the **Create App** button.



6. Name your app CollegeApp and then click **Create**.

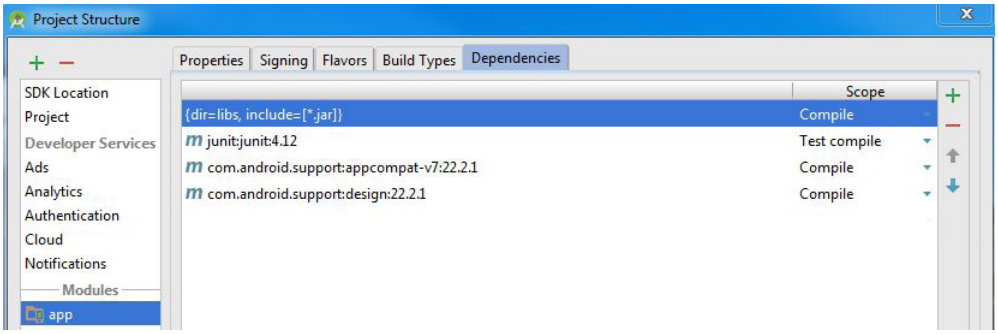


You just created space in the cloud for your app's data.

7. Prepare your Android project to interact with that service by adding the Backendless dependency:

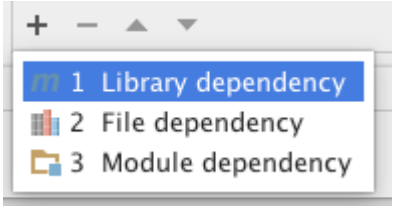
   a. Within Android Studio, select **File** > **Project Structure ...**

   b. Select **app** and then select the **Dependencies** tab as shown:

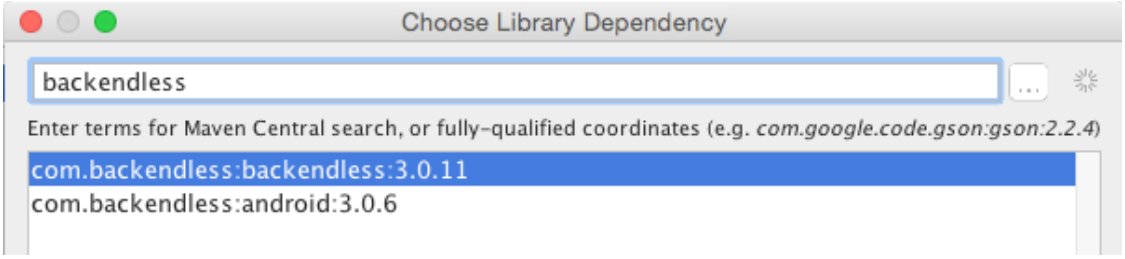


c. At the top right of the **Dependencies** tab, click the + icon.

d. Select **Library dependency** as shown:



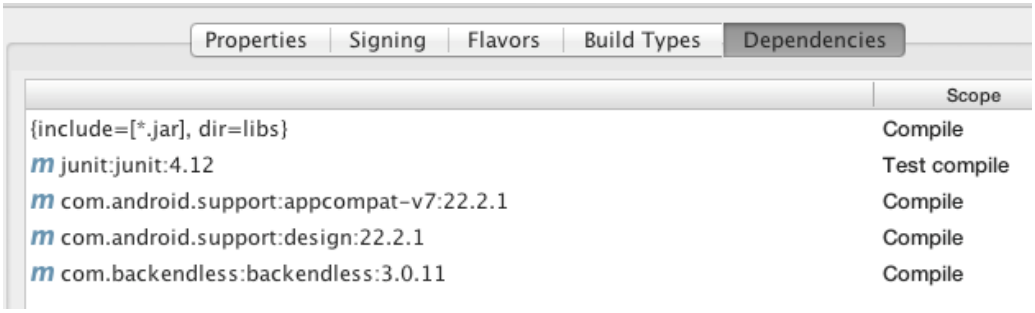
Search for "Backendless". This will brings up results similar to the following.



e. Select the version preceded by `com.backendless:backendless`. (Your version will likely be newer than the one shown.)

f. Click **OK**.

Maven, Android Studio's dependency manager, should automatically add the Backendless dependency to your project as shown below.



g. If there is another **OK** button at the bottom of the Project Structure window, click the **OK** button.

Gradle will automatically perform a project sync to incorporate the new dependency.

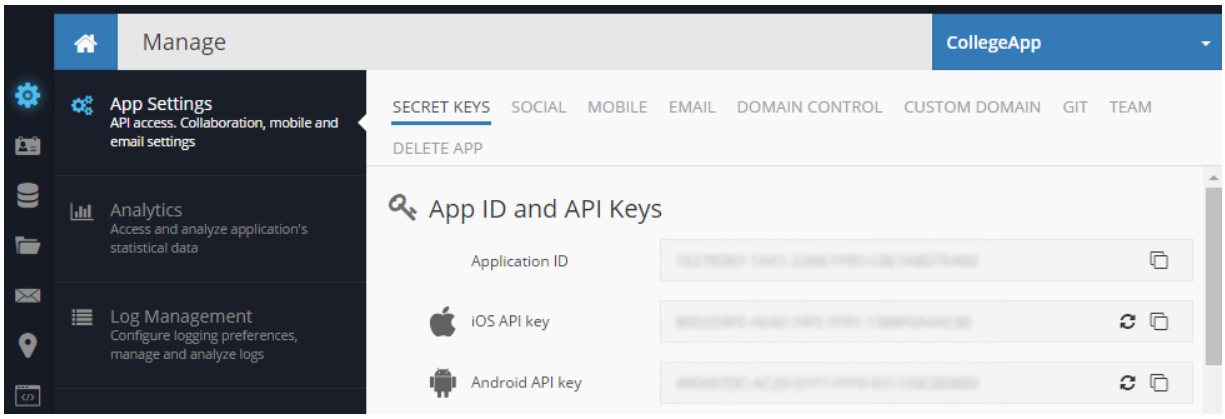
8 Open your app's manifest file and add the following permissions just after the `manifest` tag and just before the `application` tag:

```
1: <uses-permission android:name="android.permission.INTERNET" />
2: <uses-permission android:name="android.permission.ACCESS_
   NETWORK_STATE" />
```

9. Within `ApplicantActivity`, import `com.backendless.Backendless`.

   So that users can access CollegeApp's backend, the app needs your Backendless Application ID and Android API Key.

10. Return to your browser, and within the Backendless site for your app, navigate to **Manage** > **App Settings**.



11. Add the following `String` constants to your `ApplicantActivity`:

    a. `BE_APP_ID`: This constant should contain the value of your Backendless Application ID.

    b. `BE_ANDROID_API_KEY`: This constant should contain the value of your Backendless Android API key.

12. Back in Android Studio, initialize a connection to Backendless in your **onCreate** method: Add a call to the static method `initApp` of `Backendless` with parameters `this`, `BE_APP_ID`, and `BE_ANDROID_API_KEY`.

    This sets up Backendless to access the remote database from your app.

13. Check that your app is able to connect to the Backendless servers. Add the following code to your `ApplicantActivity`, using an email account of yours and your password. You need to decide where it is most appropriate to add this code.

NOTE

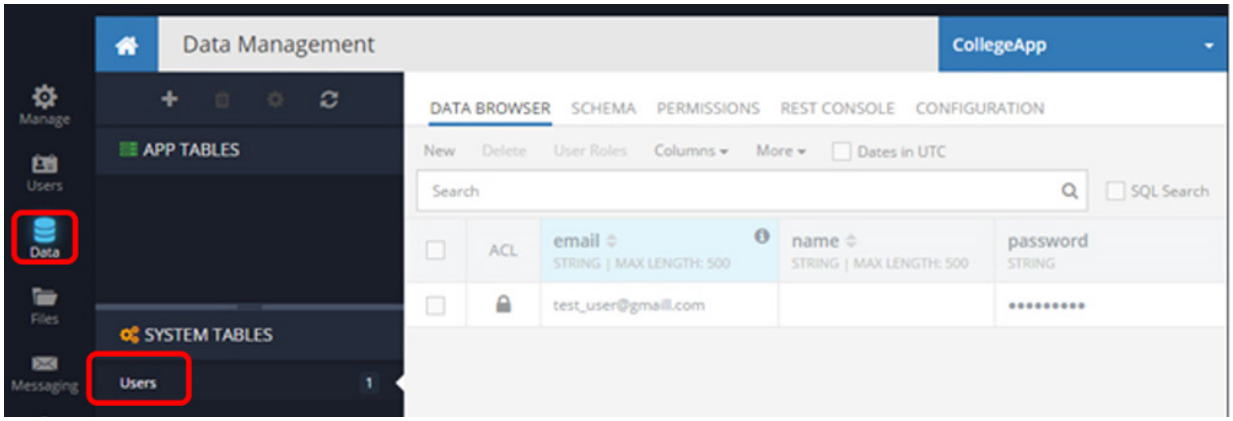
This is temporary code, so feel free to use literals in place of YOUR_EMAIL_ADDRESS and YOUR_PASSWORD.

```
BackendlessUser user = new BackendlessUser();
user.setEmail( YOUR_EMAIL_ADDRESS );
user.setPassword( YOUR_PASSWORD );

Backendless.UserService.register(user, new
AsyncCallback<BackendlessUser>(){
    @Override
    public void handleResponse(BackendlessUser backendlessUser){
        Log.i( "User", backendlessUser.getEmail() +
        " successfully registered" );
    }
    @Override
    public void handleFault(BackendlessFault backendlessFault) {
        Log.e( "Backendless registration error! ",
        backendlessFault.getMessage());
    }
});
```

a. Run your app.

If you have completed the previous steps successfully, you should see a new user with your email address in the Backendless browser interface: On the left side of the Backendless page, click the **Data** icon. You may also need to select the **Users** table in the **SYSTEM TABLES** panel:



Within moments, the email address that you used should receive an email from Backendless. (You have the ability to customize the content of this email that is sent on behalf of your app to your users, though that is outside the scope of this course.)

b. If the previous steps did not complete successfully, you should see Backendless errors in the logcat. Work with your instructor to fix the errors, either in your code or in your Backendless database.

c. If you try to run your app again after a *successful* user registration, you will see a Backendless error in the logcat, because you are trying to create an account with the same credentials. To avoid this, remove the block of code that registers a user.

To make your app data persistent, you will save your profile data to the Backendless database. You will gain some basic fluency with the Backendless API by creating a `Profile` table in Backendless. This table will correspond to the `Profile` object in the Model layer of your app.

14. Override `ProfileFragment's public void onPause()` method. As the first line of this method, call `onPause` of the `super` class.

    You will program your app to save your `Profile` information to Backendless when the `ProfileFragment` is paused. This will happen whenever the user navigates away from `ProfileFragment`.

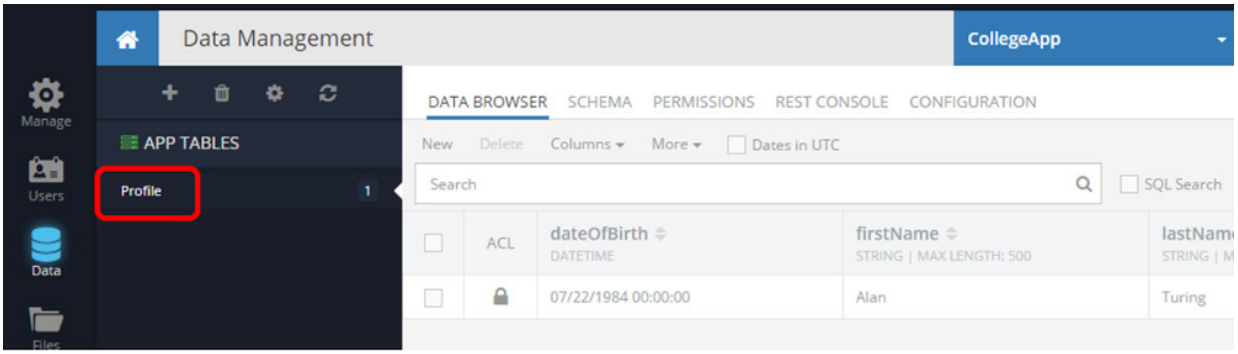
15. Add the following code to the `onPause` method after the call to `super.onPause()`.

```
Backendless.Data.of(Profile.class).save(mProfile, new
AsyncCallback<Profile>() {
    @Override
    public void handleResponse(Profile response) {
        Log.i(TAG, "Saved profile to Backendless");
    }
    public void handleFault(BackendlessFault fault) {
        Log.i(TAG, "Failed to save profile!" + fault.
        getMessage());
    }
});
```

16. Run your app: Navigate to **Profile** and then navigate to **Guardian**. This will pause the profile fragment and invoke your new `onPause` method.

17. Confirm that all is working well with the Backendless browser interface:

    a. Click the **Data** icon and then the **Profile** table in the APP TABLES panel as shown:



Backendless automatically created this Profile table including dateOfBirth, firstName, and lastName. These were the properties and getters and setters that were defined for `mProfile` when you saved it in Backendless.

Backendless also automatically created an additional column in the Profile table called objectId, and assigned a unique objectId value to your entry. In fact, Backendless creates unique objectIds for every single entry in its database. These objectIds are unique for entries across all tables; in other words, objectIds uniquely identify all data entries throughout a database.

18. To manage the Backendless objectId in `Profile`, create a `String` instance variable of the same name along with its getters and setters.

You may have already noticed that there is a problem with saving profiles on every pause of the fragment: A new `Profile` object is added to your database each time you pause `ProfileFragment`. A prolife identifies the user of your app. Your app has only one logged-in user, therefore, the database should have only one entry in the Profile table. Every pause of your app is cluttering the database with redundant data. Image if your app was used by hundreds of people for a few months, or years.

19. Back in Android Studio, add a new field to your `Profile` class called `email` with appropriate getters and setters. You will use the email address to retrieve the applicant's data.

    You *could* use the Backendless objectId to save and retrieve data, but the objectId is long, unwieldly, and hard to remember. An applicant's email address is much easier to manage.

Over the next few steps, you will save the email address to shared preferences so that it will be easily accessible by all your `Fragments` and between sessions.

NOTE

As you update your app data in Backendless, you will likely need to refresh the Dasta view to see new tables. Use the refresh icon  above APP TABLES.

20. Within `ApplicantActivity`, define a private `String` constant `MY_EMAIL_ADDRESS` to contain your email address.
21. Define a public `String` constant `EMAIL_PREF` for the key by which you will access the email address with value `"EMAIL_PREF"`.
22. Add the code below to `ApplicantActivity`.

```
1: SharedPreferences sharedPreferences =
2: this.getPreferences(Context.MODE_PRIVATE);
3: SharedPreferences.Editor editor = sharedPreferences.edit();
4: editor.putString(EMAIL_PREF, MY_EMAIL_ADDRESS);
5: editor.commit();
```

Note that in line 4, the first argument is the key, whereas the second argument is the value to associate with that key.

23 Now you need to edit `ProfileFragment`. In `onPause`, before you `save` the `Profile` in Backendless, check to see whether its `email` value is `null`. If it is, use the following code to access the shared preferences and set `mProfile's email`.

```
SharedPreferences sharedPreferences =
getActivity().getPreferences(Context.MODE_PRIVATE);
String email = sharedPreferences.getString
(ApplicantActivity.EMAIL_PREF, null);
if (mProfile.getEmail() == null) {
    mProfile.setEmail(email);
}
```

Now that `mProfile` has a well-defined `email`, you will update the object that already exists in the database instead of creating a new one.

24 Before the call to `save`, you need to find and retrieve the profile from the database. You will do this with its objectID. Recall that the objectID uniquely identifies an entry in the database. The following code will accomplish the update:

```
String whereClause = "email = '" + email + "'";
DataQueryBuilder query = DataQueryBuilder.create();
query.setWhereClause(whereClause);
Backendless.Data.of(Profile.class).find(query, new
AsyncCallback<List<Profile>>() {
    @Override
    public void handleResponse(List<Profile> profile) {
        if (!profile.isEmpty()) {
            String profileId = profile.get(0).getObjectId();
            Log.d(TAG, "Object ID: " + profileId);
            mProfile.setObjectId(profileId);
        }
    }
    @Override
    public void handleFault(BackendlessFault fault) {
         Log.e(TAG, "Failed to find profile: " + fault.
         getMessage());
    }
 });
```

Before your data will persist and be accessible when you start the app, you need to attempt to load the data when the `ProfileFragment` starts.

25 Override the `onStart` method of `ProfileFragment` and call its super class's `onStart` method.

26 Perform the same **query** in `onStart` as you did in `onPause` to find the user's profile with the following changes:

> **query**
> A request to retrieve data from a database.

a. Get the applicant's email from shared preferences. You do not need to check if it has been defined, just perform the Backendless query.

b. In `handleResponse`, if the `Profile` list is not empty, you have successfully retrieved the profile, so set the entire `mProfile` object to the profile found in Backendless: `mProfile = profile.get(0);`

c. Instead of setting `mProfile's objectId`, update the values of the `TextViews` with the values found in the database.

27 Now that you have data persistence and a one-to-one relationship in your app, delete all of the profiles in your Backendless **Profile** table. Test your app and confirm that you can do the following:

a. In your app, navigate to Profile and verify that the applicant's date of birth is shown on the button.

b. In the Backendless browser, navigate between Guardian and Profile without creating multiple profiles. Note: To see a new entry in the APP TABLES panel, you may have to refresh the Backendless browser page or navigate from and to the Data page.

Part II: Exploring Data Persistence

Following the one-to-one relationship rule, say you were to allow your user to create multiple `Guardians`, and they made one for their Mother and one for their Father.

How many instances of the `String occupation` would exist in the model layer of your app?

In the scenario described above, how many entries should there be in the occupation column of the Guardian table in the database?

Later in this lesson, you will add data persistence for family members. To save guardians (and eventually siblings) of an applicant in the database, you need a way to identify all of the family members for a particular applicant. You will do this with the applicant's email address; an applicant's email will be saved with their guardian's data. Because applicants and guardians share this data, you will make an abstract parent class for the `FamilyMember` and `Profile` classes to contain this data. (The `Guardian` is already a subclass of `FamilyMember` so it does not need to be modified.)

28. Add an abstract class called `ApplicantData` to your app. Create a `String` instance variable `email` along with its getter and setter methods. Modify the `FamilyMember` and `Profile` classes so they extend `ApplicantData`. Make the necessary changes to `Profile` to accommodate this new construct (remove references to email so that `Profile` now references **`email`** defined in `ApplicantData`).

    Recall that each entry in the Backendless database contains a unique objectId. This should also be included in `ApplicantData`; Backendless needs it to retrieve the data.

29. Move the `objectId` instance variable and its getter and setter from `Profile` to `ApplicantData`.

30. Test your app and confirm that the email address is still being saved for an applicant's Profile in Backendless.

Part III: Increased Security Risks

Backendless is a convenient tool for storing data. But it introduces a vulnerability into your app: Email, passwords, and names are stored on a server connected to the internet, in other words, "in the cloud." Data services in the cloud make app development easier and provide powerful features for an app, but this ease and power introduce risk.

What are the risks when names, emails, and passwords are stored in the cloud?

If *more* users (rather than less) use a cloud service, does it increase or decrease security? Why?

This trade-off between security and “ease of use” is a common topic in software development. Which do you prefer: security or ease of use?

CONCLUSION

1. What are the benefits of adding a back end for an application?
2. What is a Backend as a Service and how does it work? Include as much detail as you can about the infrastructure and implementation.

Activity 2.2.2 Visual Aid