LoopBack - Client SDK Documentation

Android SDK

This project provides early access to advanced or experimental functionality. It may lack usability, completeness, documentation, and robustness, and may be outdated.

However, StrongLoop supports this project: Paying customers can open issues using the StrongLoop customer support system (Zendesk). Community users, please report bugs on GitHub.

For more information, see StrongLoop Labs.

The Android SDK provides a simple Java API that enables your Android app to access a LoopBack server application. It enables you to interact with your models and data sources in a comfortable native manner instead of using clunky interfaces like `AsyncHttpClient`, `JSONObject`.

Related articles:

See also: Android SDK API docs

Download Android SDK

Please ensure you have the latest version (1.5) of the Android SDK.

If you are using Eclipse, you will have a `lib/loopback-sdk-android-x.y.z.jar` library, where x.y.z is the version.

If you are using Android Studio with gradle-based builds, then you have a dependency entry specifying the SDK version.

Getting started with the guide app

The easiest way to get started with the LoopBack Android SDK is with the LoopBack Android guide app. The guide app comes ready to compile with Android Studio and each tab in the app will guide you through the SDK features available to mobile apps.

Prerequisites

If you haven't already created your application backend, see LoopBack example app. The Android guide app will connect to the this backend sample app.

Before you start, make sure you've installed the Eclipse Android Development Tools (ADT).

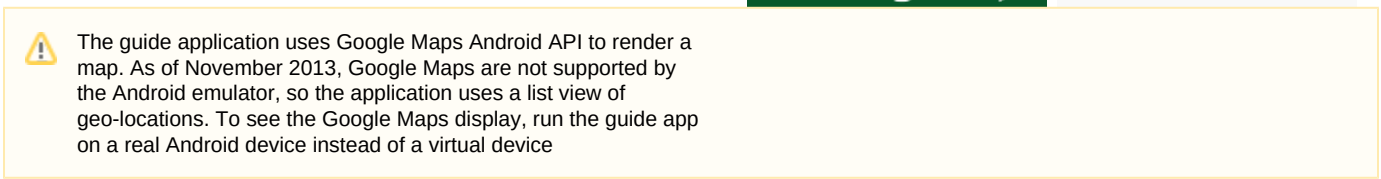
Now make sure you have the necessary SDK tools installed.

1. In ADT, choose **Window > Android SDK Manager**.

Copyright 2016 StrongLoop, an IBM company

2. Install the following if they are not already installed:
    - Tools:
        - Android SDK Platform-tools 18 or newer
        - Android SDK Build-tools 18 or newer
    - Android 4.3 (API 18)
        - SDK Platform.
3. To run the LoopBack Android guide application (see below), also install **Extras > Google Play Services**.

Before you start, make sure you have set up at least one Android virtual device: Choose **Window > Android Virtual Device Manager**. See AVD Manager for more information.



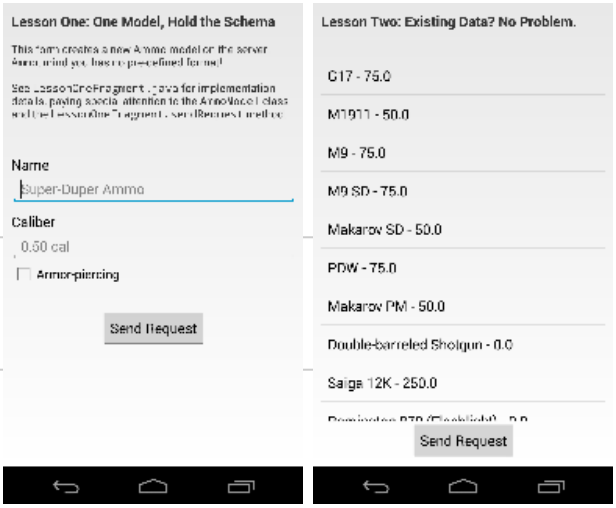
The guide application uses Google Maps Android API to render a map. As of November 2013, Google Maps are not supported by the Android emulator, so the application uses a list view of geo-locations. To see the Google Maps display, run the guide app on a real Android device instead of a virtual device

Running the LoopBack server application

Follow the instructions in Getting started with LoopBack to create the LoopBack sample backend application. You can also just clone https://github.com/strongloop/loopback-getting-started.

In the directory where you created the application, enter these commands:

```
$ cd loopback-getting-started
$ node .
```

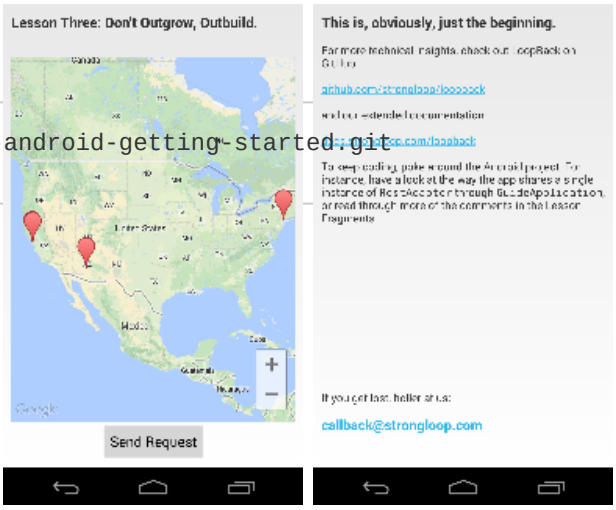


Downloading LoopBack Android Getting Started app

To get the LoopBack Android Getting Started application, you will need either the git command-line tool or a GitHub account.

To use `git`, enter this command:

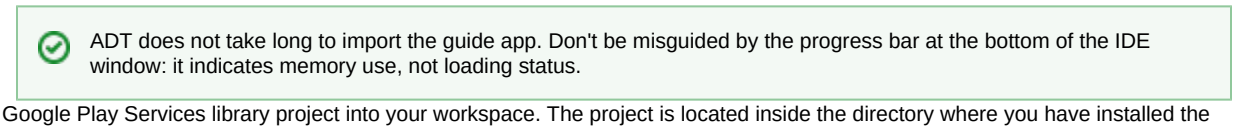
```
$ git clone git@github.com:strongloop/loopback-android-getting-started.git
```



Running the LoopBack Android Getting Started app

Follow these steps to run the LoopBack Android guide app:

1. Open ADT Eclipse.
2. Import the Loopback Guide Application to your workspace:
    a. Choose **File > Import**.
    b. Choose **Android > Existing Android Code into Workspace**.
    c. Click **Next**.
    d. Browse to the `loopback-android-getting-started/LoopbackGuideApplication` directory.
    e. Click **Finish**.



ADT does not take long to import the guide app. Don't be misguided by the progress bar at the bottom of the IDE window: it indicates memory use, not loading status.

3. Import Google Play Services library project into your workspace. The project is located inside the directory where you have installed the Android SDK.
    a. Choose **File > Import**.

   b. Choose **Android > Existing Android Code into Workspace**.
   c. Click **Next**.
   d. Browse to the `<android-sdk>/extras/google/google_play_services/libproject/google-play-services_lib` directory.
   e. Check **Copy projects into workspace**
   f. Click **Finish**.

   See Google Play Services SDK for more details.
4. Add the imported google-play-services_lib as an Android build dependency of the Guide Application.
   a. In the Package Explorer frame, select LoopbackGuideApplication
   b. Choose **File > Project Properties**
   c. Select **Android**
   d. In the Library frame, click on **Add...** and select `google-play-services_lib`
5. Obtain an API key for Google Maps Android API v2 per Getting Started instructions and enter it into `AndroidManifest.xml`.
6. Click the green **Run** button in the toolbar to run the application. Each tab (fragment) shows a different way to interact with the LoopBack server. Look at source code of fragments to see implementation details.

It takes some time for the app to initialize: Eventually, you'll see an Android virtual device window. Click the LoopBack app icon in the home screen to view the LoopBack Android guide app.

Troubleshooting

Problem: Build fails with the message `Unable to resolve target 'android-18'`.

Resolution: You need to install Android 4.3 (API 18) SDK. See Prerequisites for instructions on how to install SDK components.

If you don't want to install an older SDK and want to use the most recent one (for example, Android 4.4 API 19), follow these steps:

1. Close Eclipse ADT.
2. Edit the file `project.properties` in the `loopback-android-getting-started` directory and change the `target` property to the API version you have installed. For example: `target=android-19`.
3. Open Eclipse ADT again. The project should build correctly now.

Creating and working with your own app

If you are creating a new Android application or want to integrate an existing application with LoopBack, then follow the steps in this section.

Eclipse ADT setup

Follow these steps to add LoopBack SDK to your Eclipse project:

1. Download LoopBack Android SDK for Eclipse.
2. Extract the content of the downloaded zip file and copy the contents of the `libs` folder into the `libs` folder of your Eclipse ADT project.

Android Studio setup

1. Edit your `build.gradle` file.
2. Make sure you have `mavenCentral()` or `jcenter()` among the configured repositories (jcenter is a superset of mavenCentral). Projects created by Android Studio Beta (0.8.1 and newer) have this repository configured for you by default:

build.gradle in project root

```
repositories {
    jcenter()
}
```

3. Add `com.strongloop:loopback-sdk-android:1.5.+` to your compile dependencies:

```
dependencies {
    compile 'com.strongloop:loopback-sdk-android:1.5.+'
}
```

Working with the SDK

For the complete API documentation, see LoopBack Android API.

1. You need an adapter to tell the SDK where to find the server:

   ```
   RestAdapter adapter = new RestAdapter(getApplicationContext(),
   "http://example.com/api");
   ```

   This `RestAdapter` provides the starting point for all client interactions with the running server. It should be suffixed with "`/api`" in order for the underlying REST method calls to be resolved to your `Model`.
2. Once you have access to `adapter` (for the sake of example, assume the Adapter is available through our Fragment subclass), you can create basic `Model` and `ModelRepository` objects. Assuming you've previously created a LoopBack model named "product":

   ```
   ModelRepository productRepository = adapter.createRepository("product");
   Model pen = productRepository.createObject( ImmutableMap.of("name", "Awesome
   Pen") );
   ```

   All the normal `Model` and `ModelRepository` methods (for example, `create`, `destroy`, `findById`) are now available through `productRepository` and `pen`!
3. You can now start working with your model through the generic Model object. Continue below to learn how to extend the `Model` Java object to directly match, and thus provide the strongly-typed interface for interaction with, your own `Model`'s members. Check out the LoopBack Android API docs or create more Models with the Model generator.

Creating your own LoopBack model

Creating a subclass of `Model` enables your class to get the benefits of a Java class; for example, compile-time type checking.

Prerequisites

- **Knowledge of Java and Android app development**
- **LoopBack Android SDK** - You should have set this up when you followed one of the preceding sections.
- **Schema** - Your data schema must be defined already.

Define model class and properties

As with any Java class, the first step is to build the interface. If you leave custom behavior for later, then it's just a few property declarations and you're ready for the implementation. In this simple example, each widget has a way to be identified and a price.

```
import java.math.BigDecimal;
import com.strongloop.android.loopback.Model;

/**
 * A widget for sale.
 */
public class Widget extends Model {

  private String name;
  private BigDecimal price;

  public void setName(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }

  public void setPrice(BigDecimal price) {
    this.price = price;
  }

  public BigDecimal getPrice() {
    return price;
  }
}
```

Define model repository

The `ModelRepository` is the LoopBack Android SDK's placeholder for what in Node is a JavaScript prototype representing a specific "type" of Model on the server. In our example, this is the model exposed as "widget" (or similar) on the server:

```
var Widget = app.model('widget', {
  dataSource: "db",
  properties: {
    name: String,
    price: Number
  }
});
```

Because of this the class name (`'widget'`, above) needs to match the name that model was given on the server. If you don't have a model, see the LoopBack documentation for more information. The model *must* exist (even if the schema is empty) before it can be interacted with.

Use this to make creating Models easier. Match the name or create your own.

Since `ModelRepository` provides a basic implementation, you need only override its constructor to provide the appropriate name.

```
public class WidgetRepository extends ModelRepository<Widget> {
    public WidgetRepository() {
        super("widget", Widget.class);
    }
}
```

Add a little glue

Just as in using the guide app, you need an `RestAdapter` instance to connect to the server:

```
RestAdapter adapter = new RestAdapter("http://myserver:3000/api");
```

Remember: Replace "`http://myserver:3000/api`" with the complete URL to your server, including the the "`/api`" suffix.

Once you have that adapter, you can create our Repository instance.

```
WidgetRepository repository = adapter.createRepository(WidgetRepository.class);
```

Create and modify widgets

Now you have a `WidgetRepository` instance, you can:

Create a `Widget`:

```
Widget pencil = repository.createObject(ImmutableMap.of("name", "Pencil"));
pencil.price = new BigDecimal("1.50");
```

Save the `Widget`:

```
pencil.save(new VoidCallback() {
    @Override
    public void onSuccess() {
        // Pencil now exists on the server!
    }

    @Override
    public void onError(Throwable t) {
        // save failed, handle the error
    }
});
```

Find another `Widget`:

```
repository.findById(2, new ObjectCallback<Widget>() {
    @Override
    public void onSuccess(Widget widget) {
        // found!
    }

    public void onError(Throwable t) {
        // handle the error
    }
});
```

Remove a `Widget`:

```
pencil.destroy(new VoidCallback() {
    @Override
    public void onSuccess() {
        // No more pencil. Long live Pen!
    }

    @Override
    public void onError(Throwable t) {
        // handle the error
    }
});
```

Users and authentication

The LoopBack Android SDK provides classes that make it easy to connect an Android client app to a server application using LoopBack's authentication and authorization model:

- User: represents on the client a user instance on the server.
- UserRepository: base class implementing `ModelRepository` for the built-in User type.

See Authentication, authorization, and permissions for instructions how to enable authentication in your LoopBack server.

The Android SDK comes with a predefined implementation of UserRepository that provides loginUser and logout methods. However, you cannot use `UserRepository<User>` directly, because the Java runtime removes types from generic instances and therefore there is no way to pass the `User` class to the `UserRepository` instance created via `createRepository`. So you must create a specialized subclass:

```
package com.example.myproject;
// Optional, you can use LoopBack's User class too
public static class User extends com.strongloop.android.loopback.User {
}
public static class UserRepository·
        extends com.strongloop.android.loopback.UserRepository<User> {
    public interface LoginCallback·
        extends com.strongloop.android.loopback.UserRepository.LoginCallback<User> {
    }
    public UserRepository() {
        super("customer", null, User.class);
    }
}
```

Then use it as follows:

```
RestAdapter restAdapter = new RestAdapter("http://myserver:3000/api");
import com.example.myproject;
UserRepository userRepo = restAdapter.createRepository(UserRepository.class);
User user = userRepo.createUser("name@example.com", "password");
```

Or, to log in the user:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_login);
    final  RestAdapter  restAdapter = new RestAdapter(getApplicationContext(),
"http://myserver:3000");
    final UserRepository  userRepo =
restAdapter.createRepository(UserRepository.class);
    loginBtn = (Button) findViewById(R.id.loginButton);
    loginUsername = (EditText) findViewById(R.id.loginUsername);
    loginPassword = (EditText) findViewById(R.id.loginPassword);
    goToCreateAccountBtn = (TextView) findViewById(R.id.createAccount);
    //Login click listener
    loginBtn.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            username = loginUsername.getText().toString();
            password = loginPassword.getText().toString();
            System.out.println(username + " : " + password);
            userRepo.loginUser(username , password , new
UserRepository.LoginCallback() {
                @Override
                public void onSuccess(AccessToken token, User currentUser) {
                    Intent goToMain = new Intent(getApplicationContext(), Main.class);
                    startActivity(goToMain);
                    finish();
                    System.out.println(token.getUserId() + ":" + currentUser.getId());
                }

                @Override
                public void onError(Throwable t) {
                    Log.e("Chatome", "Login E", t);
                }
            });
        }
    });

    //Create account listener
    goToCreateAccountBtn.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            Intent goToSignup = new Intent(getApplicationContext(), Signup.class);
            startActivity(goToSignup);
            finish();
        }
    });
}
```

The `loginUser()` method returns an access token. The `RestAdapter` stores the access token in and uses it for all subsequent requests. Because it stores the value in `SharedPreferences`, it preserves the value across application restarts.

```
userRepo.logout(new VoidCallback() {
    @Override
    public void onSuccess() {
        // logged out
    }

    @Override
    public void onError(Throwable t) {
      // logout failed
    }
});
```

Accessing data of the current user

There are two methods to get the `User` object for the currently logged in user:

- `UserRepository.findCurrentUser()` - performs a request to the server to get the data of the current user and caches the response in memory. When no user is logged in, passes null to the callback.
- `UserRepository.getCachedCurrentUser()` - returns the value cached by the last call of `findCurrentUser()` or `loginUser()`.

Call `findCurrentUser` when your application starts. Then you can use the synchronous method `getCachedCurrentUser` in all your Activity classes.

Example of using the `findCurrentUser()` method:

```
userRepo.findCurrentUser(new ObjectCallback<User>() {
    @Override
    public void onSuccess(User user) {
        if (user != null) {
            // logged in
        } else {
            // anonymous user
        }
    }
});
```

Example of using the `getCachedCurrentUser()` method:

```
User currentUser = userRepo.getCachedCurrentUser();
if (currentUser != null) {
    // logged in
} else {
    // anonymous user
    // or findCurrentUser was not called yet
}
```

Extending the pre-defined User model

Most applications will need to extend the built-in User model with additional properties and methods. Consider the example of an e-shop, where the user is modeled as a `Customer`, with an additional property `address`.

models.json

```
"customer": {
  "properties": {
    "address": "string"
  },
  "options": {
    "base": "user"
  }
}
```

To access your customer model from the Android app, extend the `UserRepository` and `User` classes as you extend `ModelRepository` and `Model` when creating any other model class.

```
public class Customer extends User {
    private String address;
    public String getAddress() { return address; }
    public void setAddress(String address) { this.address = address; }
}

public class CustomerRepository extends UserRepository<Customer> {
    public interface LoginCallback extends UserRepository.LoginCallback<Customer> {
    }

     public CustomerRepository() {
        super("customer", null, Customer.class);
     }
}
```

Now you can login a customer and access the corresponding address:

```
CustomerRepository customerRepo = restAdapter.createRepository(CustomerRepository);

customerRepo.loginUser("user@example.com", "password",
    new CustomerRepository.LoginCallback() {
        @Override
        public void onSuccess(AccessToken token, Customer customer) {
            // customer was logged in
        }

       @Override
       public void onError(Throwable t) {
           // login failed
       }
   }
);

// later in one of the Activity classes
Customer current = customerRepo.getCachedCurrentUser();
if (current != null) {
    String address = current.getAddress();
    // display the address
} else {
    // you have to login first
}
```

Working with files using the Android SDK

Related articles:

See also:

- Android SDK API docs
- Storage component

Overview

The LoopBack Android SDK provides classes that enable apps to upload, store and retrieve files from a LoopBack application using the LoopBack Storage service. See Storage component for information on how to create the corresponding LoopBack server application.

The relevant classes are:

- `ContainerRepository` provides methods for creating and querying containers.
- `FileRepository` provides methods for querying existing files and uploading new files.
- `Container` represents an instance of a server-side container and provides shortcuts for some of the `FileRepository` methods.
- `File` represents an instance of a server-side file, exposes additional metadata like the public URL and provides methods for downloading the file to the Android device.

All classes are in the package `com.strongloop.android.loopback`. Since the Java platform provides a `File` class too, you may need to use fully qualified names to tell the compiler which class you want to use in your code:

- `com.strongloop.android.loopback.File` from the LoopBack Android SDK
- `java.io.File` from the Java platform

Working with containers

Creating a new container

```
ContainerRepository containerRepo =
adapter.createRepository(ContainerRepository.class);

containerRepo.create("container-name", new ObjectCallback<Container>() {
    @Override
    public void onSuccess(Container container) {
        // container was created
    }

    @Override
    public void onError(Throwable error) {
       // request failed
    }
});
```

Finding a container by name

```
containerRepo.get("container-name", new ObjectCallback<Container>() {
    @Override
    public void onSuccess(Container container) {
        // container was found
    }

    @Override
    public void onError(Throwable error) {
       // request failed
    }
});
```

Listing all containers

```
containerRepo.getAll(new ListCallback<Container>() {
    @Override
    public void onSuccess(List<Container> containers) {
        // "containers" hold all items found
    }

    @Override
    public void onError(Throwable error) {
        // request failed
    }
});
```

Working with files

All files live inside a container. The examples below assume you have a `container` object acquired by one of the methods described in the previous section.

Listing existing files

```
// same as container.getFileRepository().getAll(callback)
container.getAllFiles(new ListCallback<File>() {
    @Override
    public void onSuccess(List<File> files) {
        // process files
    }

    @Override
    public void onError(Throwable error) {
        // request failed
    }
});
```

Finding a file by name

```
// same as container.getFileRepository.get("file-name", callback)
container.getFile("file-name", new ObjectCallback<File>() {
    @Override
    public void onSuccess(File file) {
        // use the file
    }

    @Override
    public void onError(Throwable error) {
        // request failed
    }
});
```

Uploading a local file

```
java.io.File localFile = new java.io.File("path/to/file.txt");

// same as container.getFileRepository.upload(localFile, callback)
container.upload(localFile, new ObjectCallback<File>() {
    @Override
    public void onSuccess(File remoteFile) {
        // localFile was uploaded
        // call `remoteFile.getUrl()` to get its URL
    }

    @Override
    public void onError(Throwable error) {
        // upload failed
    }
});
```

Uploading in-memory content

```
String fileName = "hello.txt";
byte[] content = "Hello world".getBytes("UTF-8");
String contentType = "text/plain";

// same as container.getFileRepository().upload(fileName,...);
container.upload(fileName, content, contentType,
    new ObjectCallback<File>() {
        @Override
        public void onSuccess(File remoteFile) {
            // file was uploaded
        }

        @Override
        public void onError(Throwable error) {
            // upload failed
        }
    }
);
```

Downloading to a local file

```
File remoteFile; // obtained by one of the methods shown above
java.io.File localFile = new java.io.File("path/to/file.txt");


remoteFile.download(localFile, new VoidCallback() {
    @Override
    public void onSuccess() {
        // localFile contains the content
    }

    @Override
    public void onError(Throwable error) {
        // download failed
    }
});
```

Downloading to memory

```
File remoteFile; // obtained by one of the methods shown above

remoteFile.download(new File.DownloadCallback() {
    @Override
    public void onSuccess(byte[] content, String contentType) {
        // downloaded
    }


    @Override
    public void onError(Throwable error) {
        // download failed
    }
});
```

Removing a remote file

```
File remoteFile; // obtained by one of the methods shown above

remoteFile.delete(new Void() {
    @Override
    public void onSuccess() {
        // the file was deleted
    }

    @Override
    public void onError(Throwable error) {
        // request failed
    }
});
```

Example

For example, consider an application for submitting insurance claims. To submit a claim, one has to attach documents proving the validity of the claim, such as pictures of the damaged property.

The LoopBack server will track claims using a `Claim` model. Supporting documents will be stored in a storage service. There will be one container for every claim record. The Android application will enable users to view documents attached to a claim and to attach more documents.

See Storage component for information on setting up the server application that uses the LoopBack storage service.

Creating a new claim

To avoid extra checks further down the line, the app will create the container when the user enters a new claim in the system as shown below:

```
ContainerRepository containerRepo =
adapter.createRepository(ContainerRepository.class);

containerRepo.create(claim.getId().toString(), new ObjectCallback<Container>() {
    @Override
    public void onSuccess(Container container) {
        // container was created, save it
        activity.setContainer(container);
        // and continue to the next activity
    }

    @Override
    public void onError(Throwable error) {
       // request failed, report an error
    }
});
```

Displaying documents

To display a list of documents that are already uploaded, we need to fetch all files in the container associated with the current claim as follows:

```
activity.getContainer().getAllFiles(new ListCallback<File>() {

    @Override
    public void onSuccess(List<File> remoteFiles) {
        // populate the UI with documents
    }

    @Override
    public void onError(Throwable error) {
        // request failed, report an error
    }
}
```

To display the document, the app downloads its content and builds a `Bitmap` object that it can display on the Android device:

```
void displayDocument(File remoteFile) {

    file.download(new File.DownloadCallback() {
        @Override
        public void onSuccess(byte[] content, String contentType) {
            Bitmap image = BitmapFactory.decodeByteArray(content, 0, content.length);
            // display the image in the GUI
        }

        @Override
        public void onError(Throwable error) {
            // download failed, report an error
        }
    });
}
```

Attaching a new document

To keep this example simple, we will skip details on how to take pictures in Android (for information on this, see the Android Camera docs). Once the picture is taken, the app uploads it to the storage service and updates the list of all documents:

```
camera.takePicture(

    null, /* shutter callback */
    null, /* raw callback */
    null, /* postview callback */
    new Camera.PictureCallback() {
        /* jpeg callback */

        @Override
        void onPictureTaken(byte[] data, Camera camera) {
            // A real app would probably ask the user to provide a file name
            String fileName = UUID.randomUUID().toString() + ".jpg";

            activity.getContainer().upload(fileName, data, "image/jpeg",
                new ObjectCallback<File>() {
                    @Override
                    public void onSuccess(File remoteFile) {
                        // Update GUI - add remoteFile to the list of documents
                    }

                    @Override
                    public void onError(Throwable error) {
                        // upload failed
                    }
                }
            );
        }
    }
);
```

Push notifications using Android SDK

Related articles:

See also:

- Android SDK API docs

Overview

This article provides information on creating Android apps that can get push notifications from a LoopBack application. See Push notifications for information on creating the corresponding LoopBack server application.

To enable an Android app to receive LoopBack push notifications:

1. Setup your app to use Google Play Services.
2. On app startup, register with GCM servers to obtain a device registration ID (device token) and register the device with the LoopBack server application.
3. Configure your LoopBack application to receive incoming messages from GCM.
4. Process the notifications received.

Prerequisites

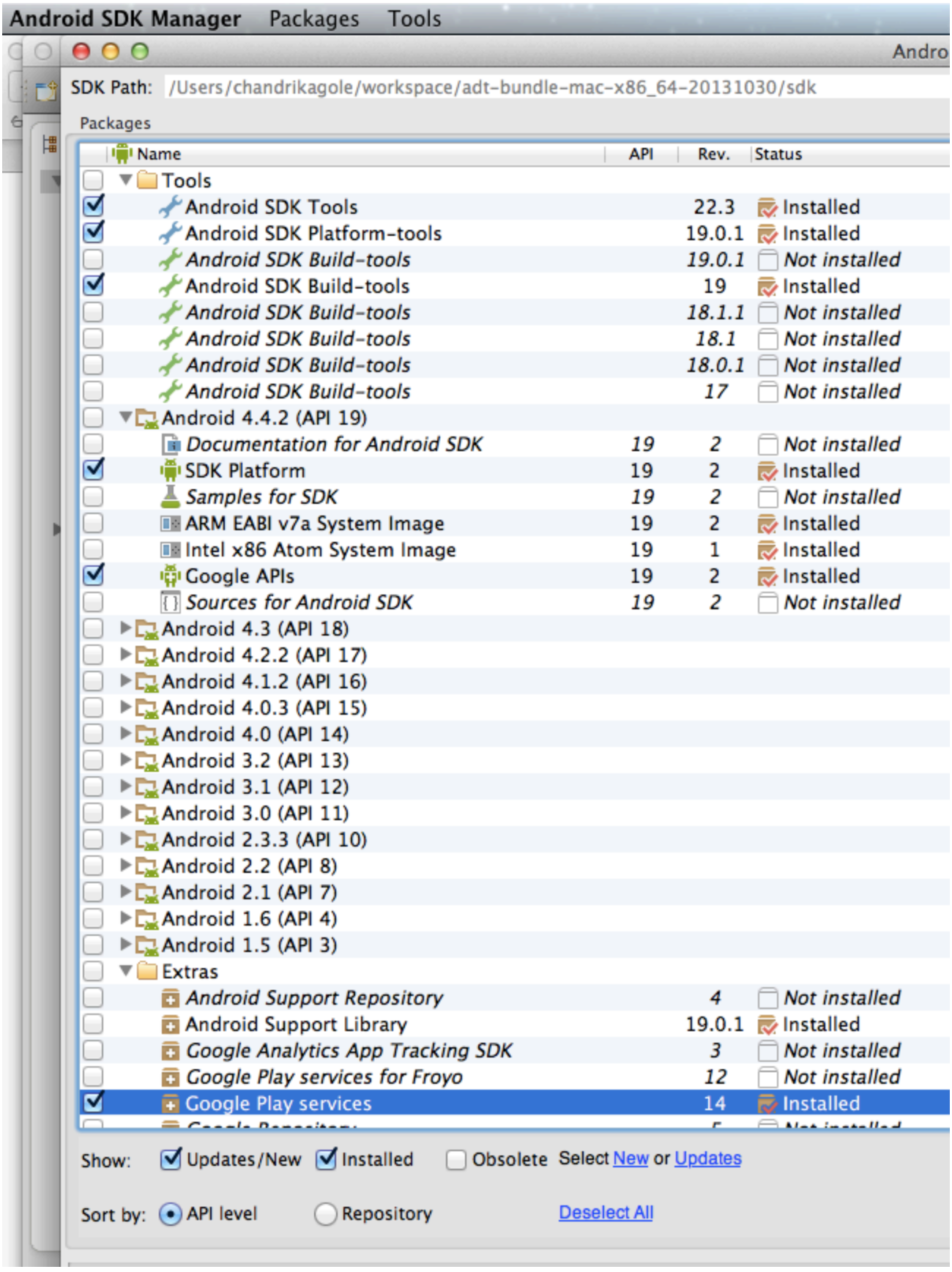
Before you start developing your application make sure you've performed all the prerequisite steps outlined in this section.

- **Download the LoopBack Android SDK**
- **Install Eclipse development tools (ADT)**

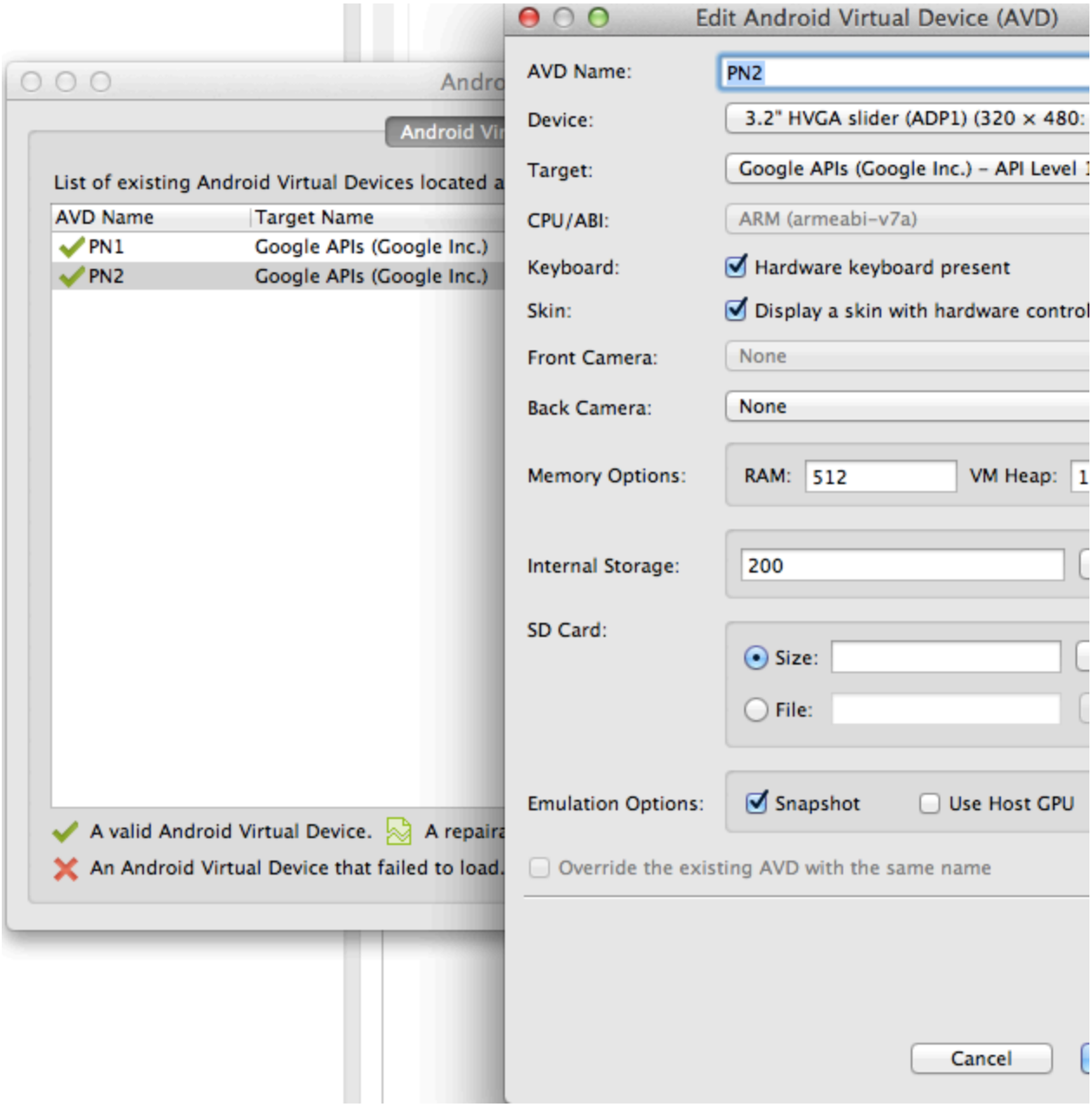
Configure Android Development Tools

Now configure Eclipse ADT as follows:

1. Open Eclipse from the downloaded ADT bundle.
2. In ADT, choose **Window > Android SDK Manager**.
3. Install the following if they are not already installed:
   - Tools:
     - Android SDK Platform-tools 18 or newer
     - Android SDK Build-tools 18 or newer
   - Android 4.3 (API 18):
     - SDK Platform.
     - Google APIs
   - Extras:
     - Google Play Services
     - Intel x86 Emulator Accelerator (HAXM)



4. Before you start, make sure you have set up at least one Android virtual device: Choose **Window > Android Virtual Device Manager**.
5. Configure the target virtual device as shown in the screenshot below. See AVD Manager for more information.



⚠ If you are using the virtual device suggested above, you must also install the ARM EABI v7a System Image SDK.

Get your Google Cloud Messaging credentials

Open the Android Developer's Guide To send push notifications to your Android app, you need to setup a Google API project and enable the Google Cloud Messaging (GCM) service.

Follow the instructions to get your GCM credentials:

1. Follow steps to create a Google API project and enable the GCM service.
2. Create an Android API key
   a. In the sidebar on the left, select **APIs & auth > Credentials**.
   b. Click **Create new key**.
   c. Select **Android key**.
   d. Enter the SHA-1 fingerprint followed by the package name, for example

      45:B5:E4:6F:36:AD:0A:98:94:B4:02:66:2B:12:17:F2:56:26:A0:E0;com.example
      NOTE: Leave the package name as "com.example" for the time being.

3. You also have to create a new server API key that will be used by the LoopBack server:

a. Click **Create new key**.
b. Select **Server key**.
c. Leave the list of allowed IP addresses empty for now.
d. Click **Create**.
e. Copy down the API key. Later you will use this when you configure the LoopBack server application.

Install and run LoopBack Push Notification app

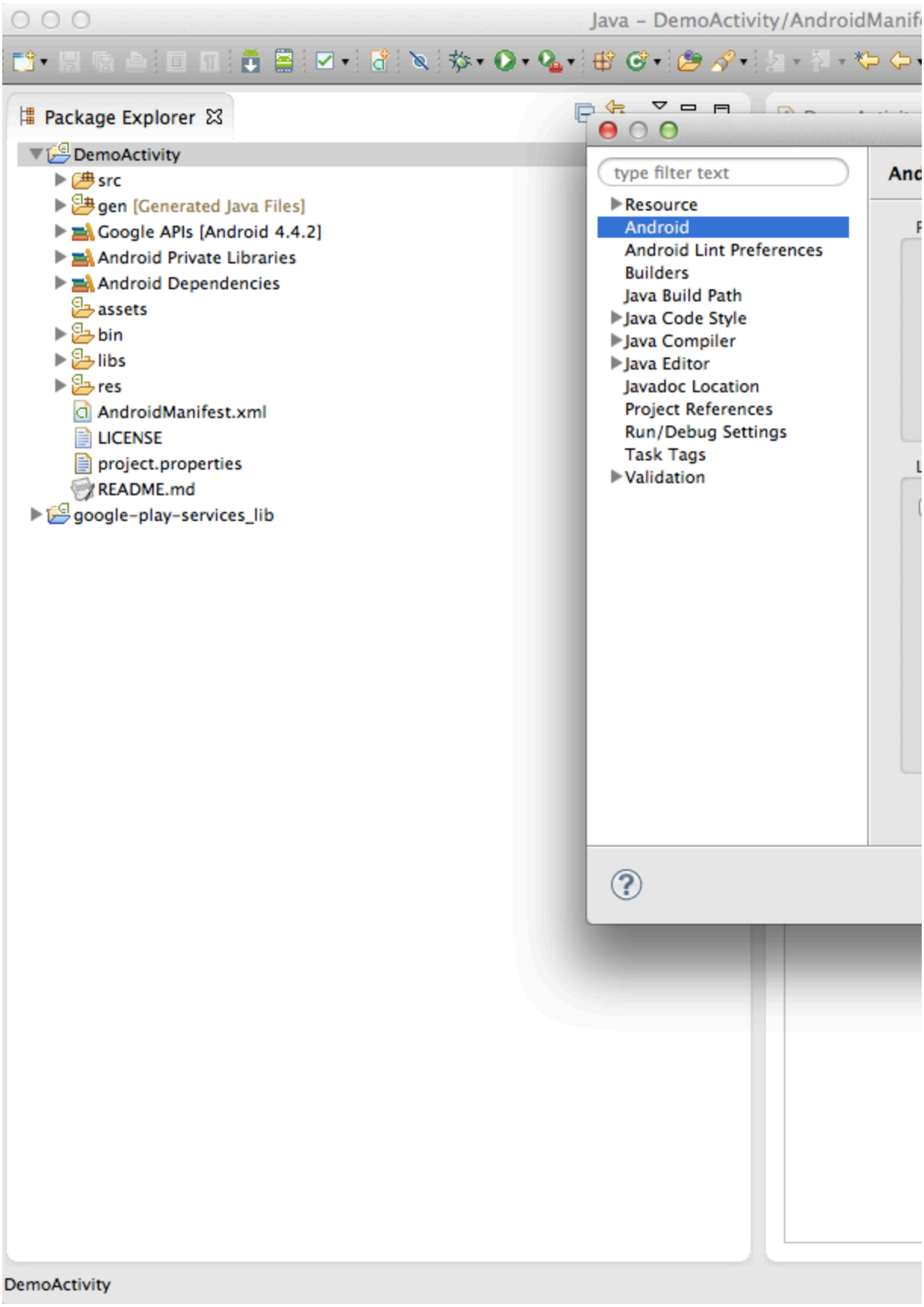
If you want to use the sample Android client app, download the Push Notification Example Android app. Then follow these steps to run the app:

1. Open ADT Eclipse.
2. Import the push notification application to your workspace:
   a. Choose **File > Import**.
   b. Choose **Android > Existing Android Code into Workspace**.
   c. Click **Next**.
   d. Browse to the example Android app you just downloaded.
   e. Click **Finish**.

   > ⚠ ADT does not take long to import the guide app. Don't be misguided by the progress bar at the bottom of the IDE window: it indicates memory use, not loading status.

3. Import Google Play Services library project into your workspace. The project is located inside the directory where you have installed the Android SDK.
   a. Choose **File > Import**.
   b. Choose **Android > Existing Android Code into Workspace**.
   c. Click **Next**.
   d. Browse to the `<android-sdk>/extras/google/google_play_services/libproject/google-play-services_lib` directory.
   e. Check **Copy projects into workspace**
   f. Click **Finish**.

   See Google Play Services SDK for more details.
4. Add the imported google-play-services_lib as an Android build dependency of the push notification application.
   a. In the Package Explorer frame in Eclipse, select the push notification application.
   b. Choose **File > Properties**.
   c. Select **Android**.
   d. In the Library frame, click on **Add...** and select `google-play-services_lib`.
   e. Also under Project Build Target, set the target as Google APIs.



5. Edit `src/com/google/android/gcm/demo/app/DemoActivity.java`.
   - Set SENDER_ID to the project number from the Google Developers Console you created earlier in Get your Google Cloud Messaging credentials.
6. Go back to the https://cloud.google.com/console/project and edit the Android Key to reflect your unique application ID. Set the value of **A**

ndroid applications to something like this:

Android applications	XX:XX:XX:XX:XX:XX:XX:XX:XX:X LOOPBACK_APP_ID X:XX:XX:XX:XX:XX:XX:XX:XX;com.google.android.gcm.demo.app.DemoApplication

7. Run the LoopBack server application you set up earlier. If you didn't set the appName in the server application's `config.js` earlier, do it now.
   Set it to "**com.google.android.gcm.demo.app.DemoActivity**".
8. Click the green **Run** button in the toolbar to run the application. Run it as an Android application. You will be prompted to select the target on which to run the application. Select the AVD you created earlier.

It may take several minutes to launch your application and the Android virtual device the first time.

Due to a known issue with Google Play Services, you must download and import an older version of Google Play services.

1. Download https://dl-ssl.google.com/android/repository/google_play_services_3225130_r10.zip
2. Extract the zip file.
3. In Eclipse ADT, right-click on your project and choose **Import**...
4. Choose **Existing Android Code into Workspace** then click Next.
5. Click **Browse**...
6. Browse to the `google-play-services/libproject/google-play-services_lib/` directory created when you extracted the zip file and select it in the dialog box.
7. Click **Finish**.

You must also update `AndroidManifest.xml` as follows:

1. In Eclipse ADT, browse to DemoActivity/AndroidManifest.xml.
2. Change the line
   `<meta-data android:name="com.google.android.gms.version" android:value="@integer/google_play_services_version"/>`
   to
   `<meta-data android:name="com.google.android.gms.version" android:value="4030500”/>`
3. Save the file.

Configure GCM push settings in your server application

Add the following key and value to the push settings of your application:

```
{
  gcm: {
    serverApiKey: "server-api-key"
  }
}
```

Replace `server-api-key` with the API key you obtained in Get your Google Cloud Messaging credentials.

Prepare your own Android project

Follow the instructions in Android SDK documentation to add LoopBack Android SDK to your Android project.

Follow the instructions in Google's Implementing GCM Client guide for setting up Google Play Services in your project.

To use push notifications, you must install a compatible version of the Google APIs platform. To test your app on the emulator, expand the directory for Android 4.2.2 (API 17) or a higher version, select **Google APIs**, and install it. Then create a new AVD with Google APIs as the platform target. You must install the package from the SDK manager. For more information, see Set Up Google Play Services.

Check for Google Play Services APK

Applications that rely on the Google Play Services SDK should always check the device for a compatible Google Play services APK before using Google Cloud Messaging.

For example, the following code checks the device for Google Play Services APK by calling `checkPlayServices()` if this method returns true, it proceeds with GCM registration. The `checkPlayServices()` method checks whether the device has the Google Play Services APK. If it doesn't, it displays a dialog that allows users to download the APK from the Google Play Store or enables it in the device's system settings.

```
@Override
public void onCreate(final Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    if (checkPlayServices()) {
        updateRegistration();
    } else {
        Log.i(TAG, "No valid Google Play Services APK found.");
    }
}
private boolean checkPlayServices() {
    int resultCode = GooglePlayServicesUtil.isGooglePlayServicesAvailable(this);
    if (resultCode != ConnectionResult.SUCCESS) {
        if (GooglePlayServicesUtil.isUserRecoverableError(resultCode)) {
            GooglePlayServicesUtil.getErrorDialog(resultCode, this,
                    PLAY_SERVICES_RESOLUTION_REQUEST).show();
        } else {
            Log.i(TAG, "This device is not supported.");
            finish();
        }
        return false;
    }
    return true;
}
```

Create LocalInstallation

Once you have ensured the device provides Google Play Services, the app can register with GCM and LoopBack (for example, by calling a method such as `updateRegistration()` as shown below). Rather than register with GCM every time the app starts, simply store and retrieve the registration ID (device token). The `LocalInstallation` class in the LoopBack SDK handles these details for you.

For more information on LocallInstallation, see Working with the LocalInstallation class.

The example `updateRegistration()` method does the following:

- Lines 3 - 4: get a reference to the shared `RestAdapter` instance.
- Line 5: Create an instance of `LocalInstallation`.
- Line 13: Subscribe to topics.
- Lines 15-19: Check if there is a valid GCM registration ID. If so, then save the installation to the server; if not, get one from GCM and then save the installation.

```
private void updateRegistration() {

    final DemoApplication app = (DemoApplication) getApplication();
    final RestAdapter adapter = app.getLoopBackAdapter();
    final LocalInstallation installation = new LocalInstallation(context, adapter);

    // Substitute the real ID of the LoopBack application as created by the server
    installation.setAppId("loopback-app-id");

    // Substitute a real ID of the user logged in to the application
    installation.setUserId("loopback-android");

    installation.setSubscriptions(new String[] { "all" });

    if (installation.getDeviceToken() != null) {
        saveInstallation(installation);
    } else {
        registerInBackground(installation);
    }
}
```

Register with GCM if needed

In the following code, the application obtains a new registration ID from GCM. Because the `register()` method is blocking, you must call it on a background thread.

```
private void registerInBackground(final LocalInstallation installation) {
    new AsyncTask<Void, Void, Exception>() {
        @Override
        protected Exception doInBackground(final Void... params) {
            try {
                GoogleCloudMessaging gcm = GoogleCloudMessaging.getInstance(this);
                // substitute 12345 with the real Google API Project number
    final String regid = gcm.register("12345");
                installation.setDeviceToken(regid);
                return null;
            } catch (final IOException ex) {
                return ex;
                // If there is an error, don't just keep trying to
                // register.
                // Require the user to click a button again, or perform
                // exponential back-off.
            }
        }
        @Override
        protected void onPostExecute(final Exception error) {
            if (err != null) {
                Log.e(TAG, "GCM Registration failed.", error);
            } else {
                saveInstallation(installation);
            }
        }
    }.execute(null, null, null);
}
```

Register with LoopBack server

Once you have all Installation properties set, you can register with the LoopBack server. The first run of the application should create a new Installation record, subsequent runs should update this existing record. The LoopBack Android SDK handles the details. Your code just needs to call `save()`.

```
void saveInstallation(final LocalInstallation installation) {
    installation.save(new Model.Callback() {
        @Override
        public void onSuccess() {
            // Installation was saved.
            // You can access the id assigned by the server via
            //   installation.getId();
        }
        @Override
        public void onError(final Throwable t) {
            Log.e(TAG, "Cannot save Installation", t);
        }
    });
}
```

Handle received notifications

Android apps handle incoming notifications in the standard way; LoopBack does not require any special changes. For more information, see the section "Receive a message" of Google's Implementing GCM Client guide.

Troubleshooting

When running your app in the Eclipse device emulator, you may encounter the following error:

```
Google Play services, which some of your applications rely on, is not supported by your device. Please
contact the manufacturer for assistance.
```

To resolve this, install a compatible version of the Google APIs platform. See Prepare your Android project for more information.

Working with the LocalInstallation class

- Overview: Android LocalInstallation class
- Associating installations with authenticated users

Related articles:

See also: Android SDK API docs

Overview: Android LocalInstallation class

The `LocalInstallation` class is not a client mirror of the server `Installation` Model. Instead, it wraps the `Installation` model into an API that makes it easy to implement the correct client semantics.

1. When `LocalInstallation.save()` is called by the application for the first time, a new `Installation` record is created on the server. The data are cached in application's `SharedPreferences`, thus all subsequent calls of `save()` update this existing record. That way there is exactly one `Installation` record for each instance of the application (each phone running the app).
2. It is ok to create a new `LocalInstallation` instance whenever you need to update any of the `Installation` properties. Since the data is cached in `SharedPreferences`, all `LocalInstallation` objects are initialized from the same data at the creation time. Just don't forget to call `save` to persist your changes both on the server and in the local storage.

Associating installations with authenticated users

1. By default, `userId` is `null`, which means the user has not logged in yet (has not created an account yet).
2. After a successful login, call `LocalInstallation.setUserId()` to update the user relation and save the changes.

```
userRepository.loginUser(email, password, new
UserRepository<User>.LoginCallback() {
    @Override
    public void onSuccess(AccessToken token, User currentUser) {
        final LocalInstallation installation = new LocalInstallation(context,
adapter);
        installation.setUserId(currentUser.getId());
        installation.save(/* callback */);
    }

    @Override
    public void onError(Throwable t) {
        // handle the error
    }
);
```

3. After a logout, set `userId` back to `null` in order to flag the installation as anonymous.

```
userRepository.logout(new UserRepository<User>.VoidCallback() {
    @Override
    public void onSuccess() {
        final LocalInstallation installation = new LocalInstallation(context,
adapter);
        installation.setUserId(null);
        installation.save(/* callback */);
    }

    @Override
    public void onError(Throwable t) {
        // handle the error
    }
);
```

Alternatively, you can use the `status` property to flag the installations where the user was logged out. This way it is possible to tell which user was the last one logged on the device:

1. A new installation has `userId: null` and `status: "Active"`.
2. Login updates the `userId` with a non-empty id value and sets `status: "Active"`.
3. Logout keeps the `userId` value but changes the `status` to a differed value, e.g. `"LoggedOut"`.
4. Subsequent login updates the `userId` to the new id value and sets `status` back to `"Active"`.

iOS SDK

StrongLoop Labs

This project provides early access to advanced or experimental functionality. It may lack usability, completeness, documentation, and robustness, and may be outdated.

However, StrongLoop supports this project: Paying customers can open issues using the StrongLoop customer support system (Zendesk). Community users, please report bugs on GitHub.

For more information, see StrongLoop Labs.

See also the LoopBack iOS SDK API reference.

Overview

The LoopBack iOS SDK eliminates the need to use the clunky `NSURLRequest` and similar interfaces to interact with a LoopBack-based backend. Instead, interact with your models and data sources in a comfortable, first-class, native manner.

Download iOS SDK

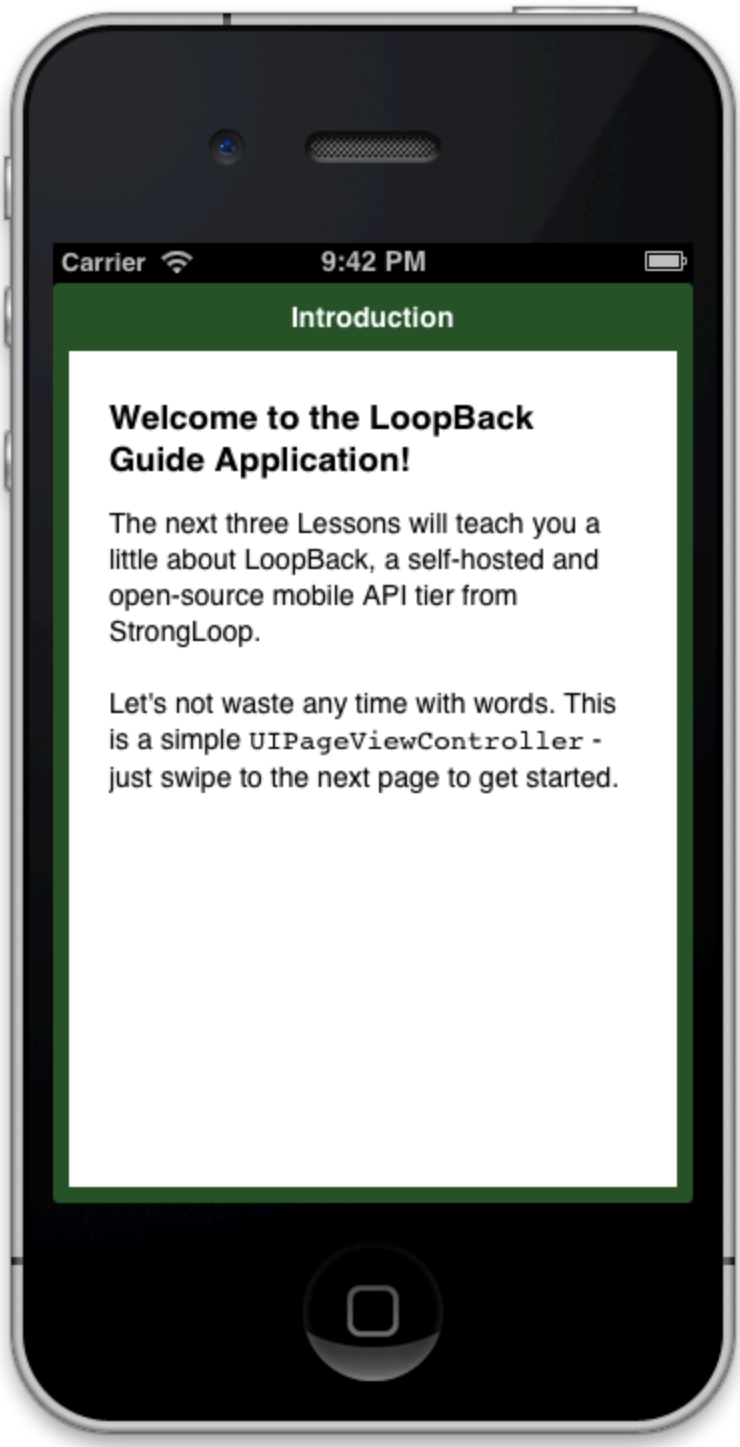
Prerequisites

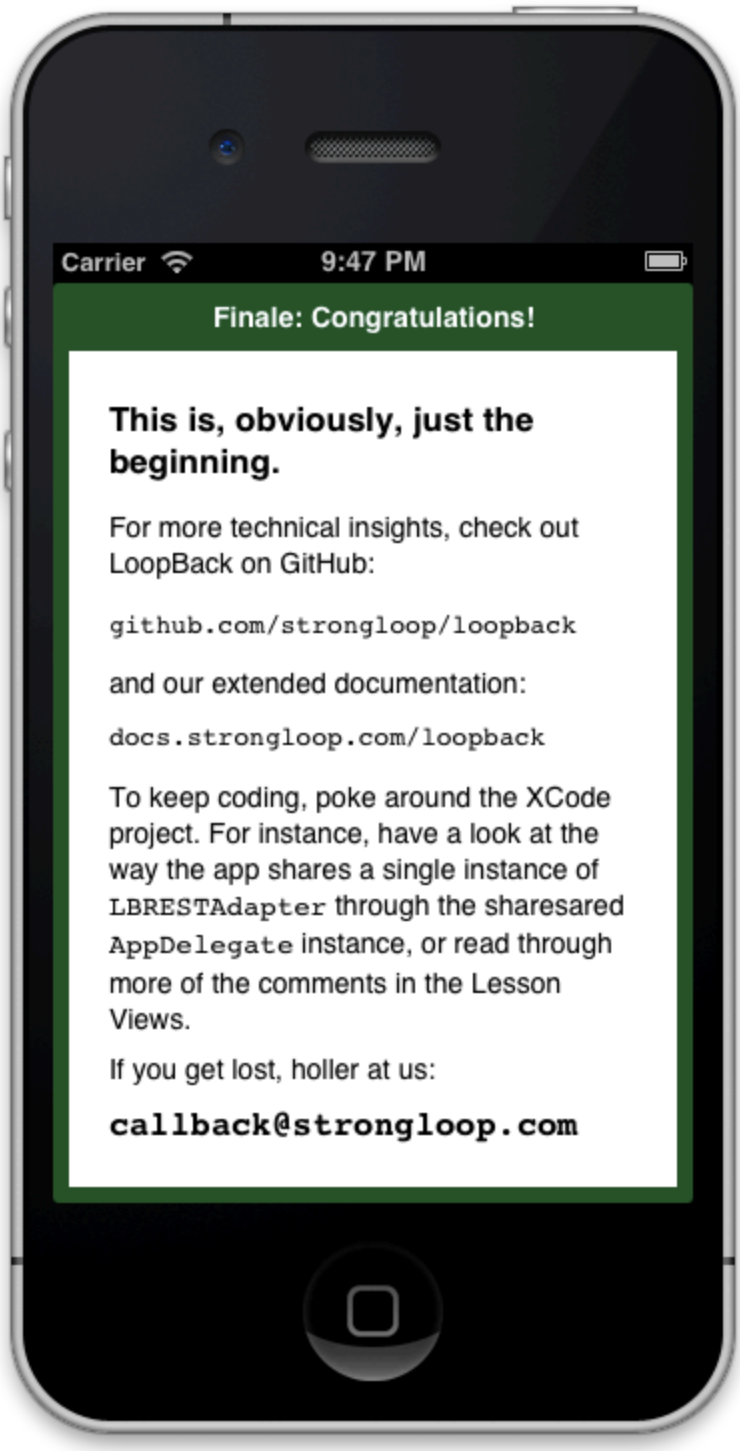
- Knowledge of Objective-C and iOS App Development
- Mac OSX with Xcode 4.6 or higher
- For on-device testing, an iOS device with iOS 5 or higher
- A LoopBack-powered server application.
- App schema. Explaining the type of data to store and why it is outside the scope of this guide, being tightly coupled to your application's needs.

The LoopBack iOS guide app

The easiest way to get started with the LoopBack iOS SDK is with the LoopBack iOS guide app. The guide app comes ready to compile with XCode, and each tab in the app guides you through the features available to mobile apps through the SDK. Here are some representative screenshots:

Copyright 2016 StrongLoop, an IBM company





From your usual projects directory:

1. Download the LoopBack guide application to your local machine from GitHub:

```
$ git clone git@github.com:strongloop/loopback-ios-getting-started.git
```

2. Open the Xcode project downloaded as a part of the Guide Application's Git repository.

```
$ cd loopback-ios-getting-started\LoopBackGuideApplication
$ open LoopBackGuideApplication.xcodeproj
```

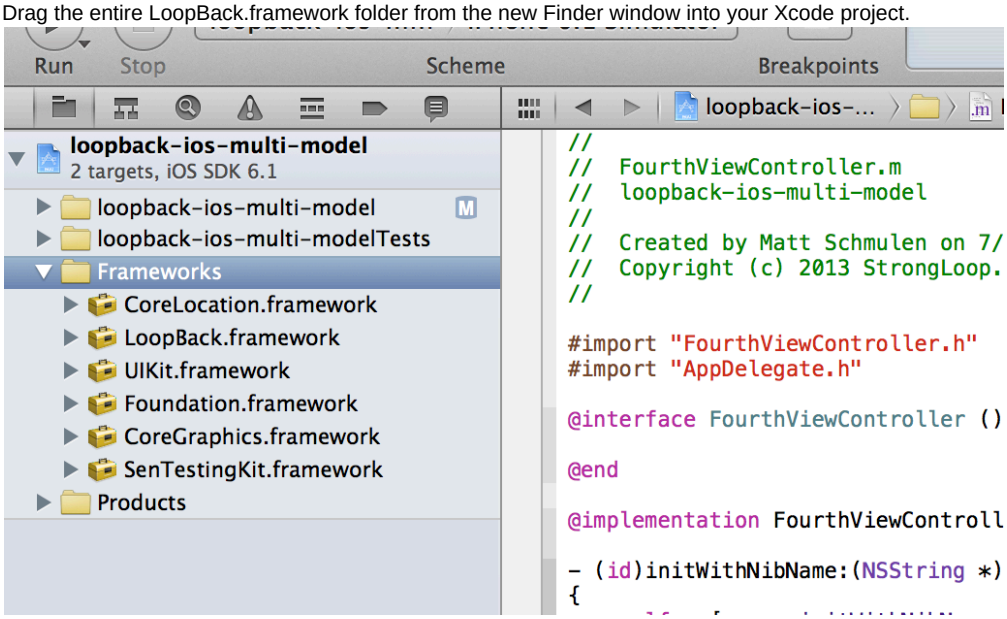
3. Run the application from Xcode (Command+R by default) and follow the instructions on each tab. Popup dialogs in the application will ask you to uncomment various code blocks in each ViewController illustrating how to use the LoopBack SDK to interact with models stored on the server.

Getting Started with the iOS SDK

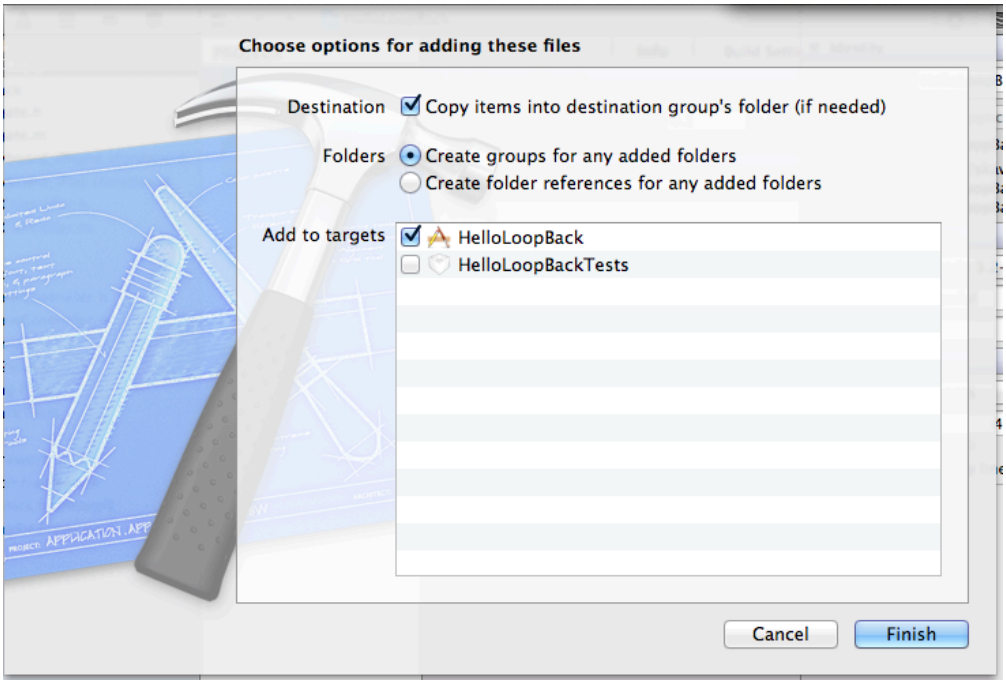
If you are creating a new iOS application or want to integrate an existing application with LoopBack, use the LoopBack SDK directly (LoopBack.framework), independent of the guide application.

Follow these steps:

1. Open the Xcode project you want to use with LoopBack, or create a new one.
2. Drag the entire LoopBack.framework folder from the new Finder window into your Xcode project.

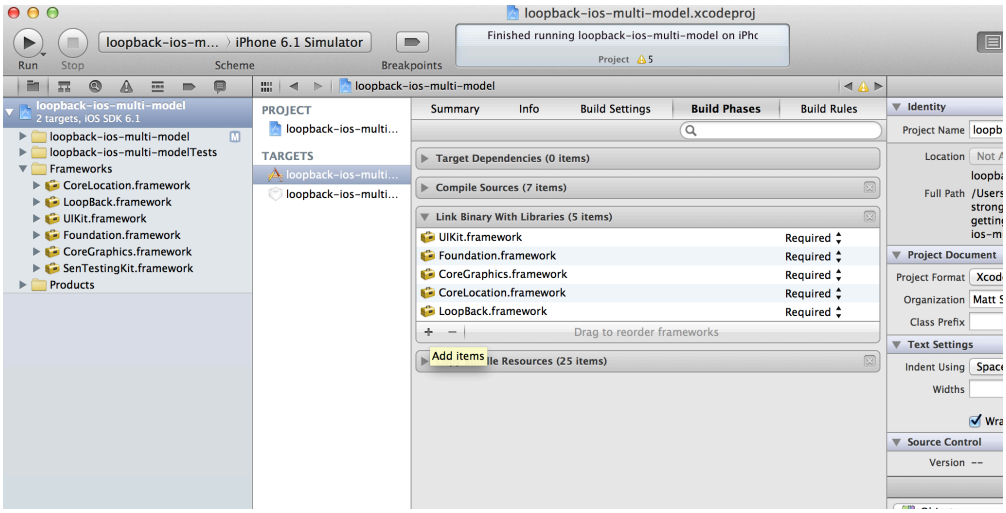


Important:Make sure to select "Copy items into destination group's folder". This places a copy of the SDK within your application's project folder.



3. Verify LoopBack is included in the list of iOS Frameworks to link against your binary. In your Project settings, check the 'Link with Binaries' section under the 'Build Phases' tab. If it's missing, add it directly by clicking the '+' button and selecting LoopBack.framework.

> ⚠ If **LoopBack.framework** isn't displayed in the list, try the previous step again; Xcode didn't create the copy it was supposed to create.



4. Import the LoopBack.h header into your application just as you would Foundation/Foundation.h. Type this line:

```
#import <LoopBack/LoopBack.h>
```

5. You need an `Adapter` to tell the SDK where to find the server. Enter this code:

```
LBRESTAdapter *adapter = [LBRESTAdapter adapterWithURL:[NSURL
URLWithString:@"http://example.com"]];
```

This `LBRESTAdapter` provides the starting point for all our interactions with the running and anxiously waiting server.

Once we have access to `adapter` (for the sake of example, we'll assume the Adapter is available through our AppDelegate), we can create basic `LBModel` and `LBModelRepository` objects. Assuming we've previously created a model named "product":

```
LBRESTAdapter *adapter = [[UIApplication sharedApplication] delegate].adapter;
LBModelRepository *productReposiory = [adapter
repositoryWithModelName:@"products"];
LBModel *pen = [Product modelWithDictionary:@{ "name": "Awesome Pen" }];
```

All the normal `LBModel` and `LBModelRepository` methods (for example, `create`, `destroy`, and `findById`) are now available through `Product` and `pen`!

6. Go forth and develop! Check out the API docs or create more Models with the slc command-line tool.

Creating a sub-class of LBModel

Creating a subclass of LBModel enables you to get the benefits of an Objective-C class (for example, compile-time type checking).

Model interface and properties

As with any Objective-C class, the first step is to build your interface. If we leave any custom behavior for later, then it's just a few `@property` de clarations and we're ready for the implementation.

```
/** * A widget for sale. */
@interface Widget : LBModel // This is a subclass, after all.

// Being for sale, each widget has a way to be identified and an amount of
// currency to be exchanged for it. Identifying the currency to be exchanged is
// left as an uninteresting exercise for any financial programmers reading this.
@property (nonatomic, copy) NSString *name;
@property (nonatomic) NSNumber *price;

@end
```

Model implementation

Since we've left custom behavior for later, just leave this here.

```
@implementation Widget
@end
```

Repository interface

The `LBModelRepository` is the LoopBack iOS SDK's placeholder for what in Node is a JavaScript prototype representing a specific "type" of Model on the server. In our example, this would be the model exposed as "widget" (or similar) on the server:

```
var Widget = app.model('widget', {
  dataSource: "db",
  properties: {
    name: String,
    price: Number
  }
});
```

Because of this the repository class name (`'widget'`, above) needs to match the name that model was given on the server.

If you haven't created a model yet, see Defining models. The model *must* exist (even if the schema is empty) before your app can interact with it.

Use this to make creating Models easier. Match the name or create your own.

Since `LBModelRepository` provides a basic implementation, we only need to override its constructor to provide the appropriate name.

```
@interface WidgetRepository : LBModelRepository

+ (instancetype)repository;

@end
```

Repository implementation

Remember to use the right name:

```
@implementation WidgetRepository

+ (instancetype)repository {
    return [self repositoryWithClassName:@"widget"];
}

@end
```

A little glue

Just as you did in Getting started, you'll need an `LBRESTAdapter` instance to connect to our server:

```
LBRESTAdapter *adapter = [LBRESTAdapter adapterWithURL:[NSURL
URLWithString:@"http://myserver:3000"]];
```

Remember: Replace `"http://myserver:3000"` with the complete URL to your server.

Once you have that adapter, you can create a repository instance.

```
WidgetRepository *repository = (WidgetRepository *)[adapter
repositoryWithModelClass:[WidgetRepository class]];
```

Using the repository instance

Now that you have a `WidgetRepository` instance, you can create, save, find, and delete widgets, as illustrated below.

Create a Widget:

```
Widget *pencil = (Widget *)[repository modelWithDictionary:@{ @"name": @"Pencil",
@"price": @1.50 }];
```

Save a Widget:

```
[pencil saveWithSuccess:^{
                    // Pencil now exists on the server!
                }
                failure:^(NSError *error) {
                    NSLog("An error occurred: %@", error);
                }];
```

Find another Widget:

```
[repository findWithId:@2
              success:^(LBModel *model) {
                  Widget *pen = (Widget *)model;
              }
              failure:^(NSError *error) {
                  NSLog("An error occurred: %@", error);
              }];
```

Remove a Widget:

```
[pencil destroyWithSuccess:^{
                       // No more pencil. Long live Pen!
                   }
                   failure:^(NSError *error) {
                       NSLog("An error occurred: %@", error);
                   }];
```

iOS SDK (version 1.3)

See also the LoopBack iOS SDK API reference.

Overview

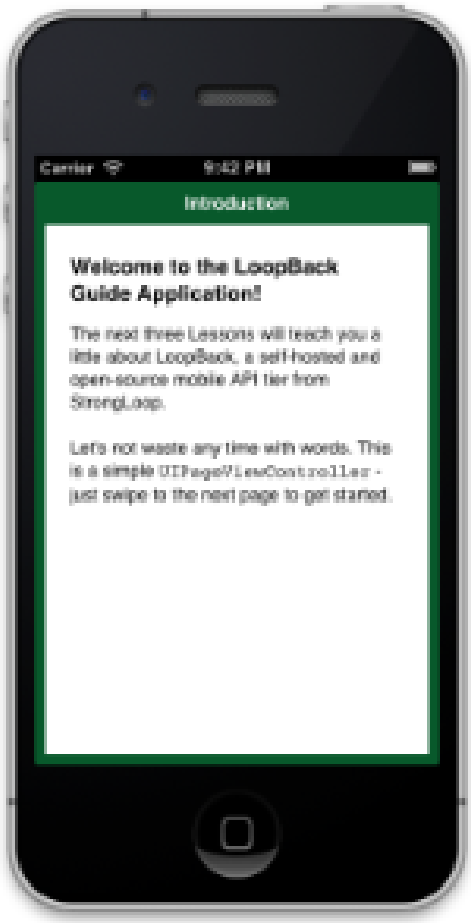
The LoopBack iOS SDK eliminates the need to use the clunky `NSURLRequest` and similar interfaces to interact with a LoopBack-based backend. Instead, interact with your models and data sources in a comfortable, first-class, native manner.

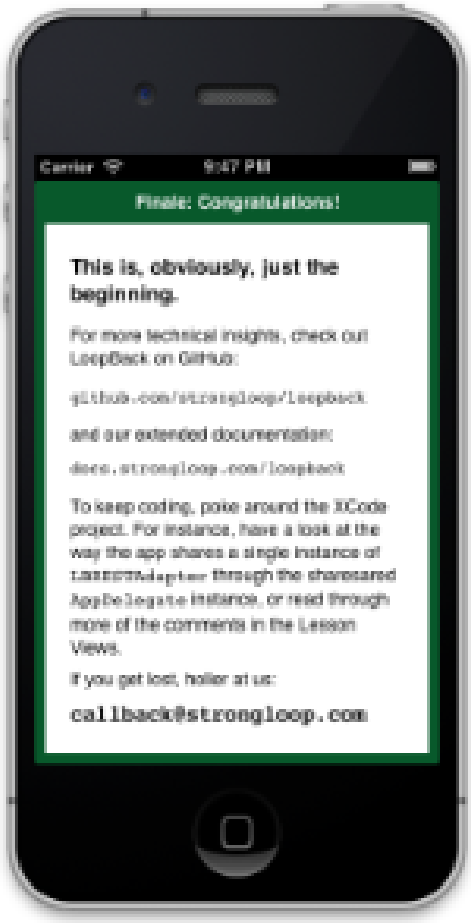
Prerequisites

- Knowledge of Objective-C and iOS App Development
- Mac OSX with Xcode 4.6 or higher
- For on-device testing, an iOS device with iOS 5 or higher
- A LoopBack-powered server application.
- App schema. Explaining the type of data to store and why it is outside the scope of this guide, being tightly coupled to your application's needs.

The LoopBack iOS guide app

The easiest way to get started with the LoopBack iOS SDK is with the LoopBack iOS guide app. The guide app comes ready to compile with XCode, and each tab in the app guides you through the features available to mobile apps through the SDK. Here are some representative screenshots:





From your usual projects directory:

1. Download the LoopBack guide application to your local machine from GitHub:

```
$ git clone git@github.com:strongloop/loopback-ios-getting-started.git
```

2. Open the Xcode project downloaded as a part of the Guide Application's Git repository.

```
$ cd loopback-ios-getting-started\LoopBackGuideApplication
$ open LoopBackGuideApplication.xcodeproj
```

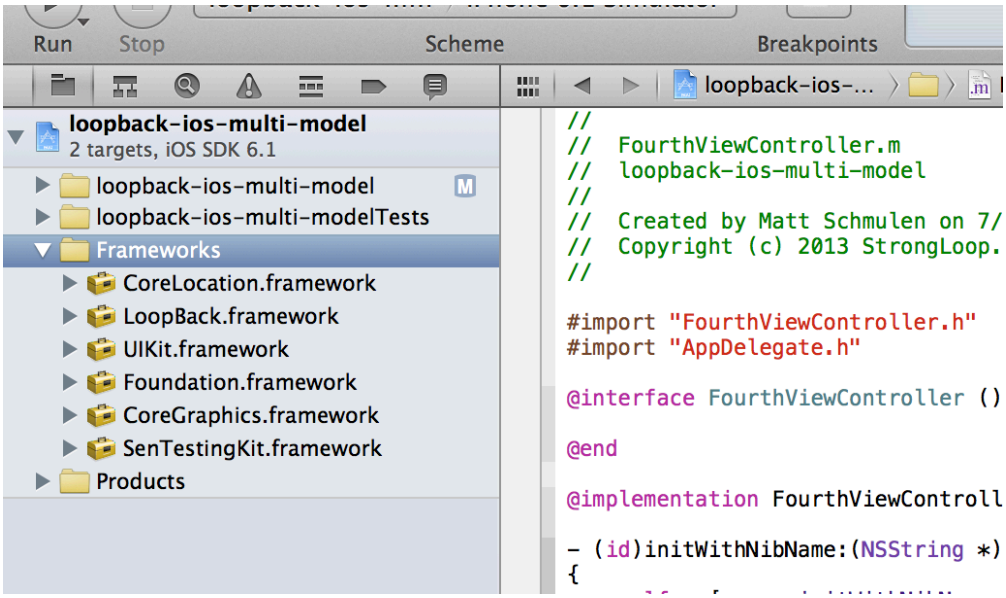
3. Run the application from Xcode (Command+R by default) and follow the instructions on each tab. Popup dialogs in the application will ask you to uncomment various code blocks in each ViewController illustrating how to use the LoopBack SDK to interact with models stored on the server.

Getting Started with the iOS SDK

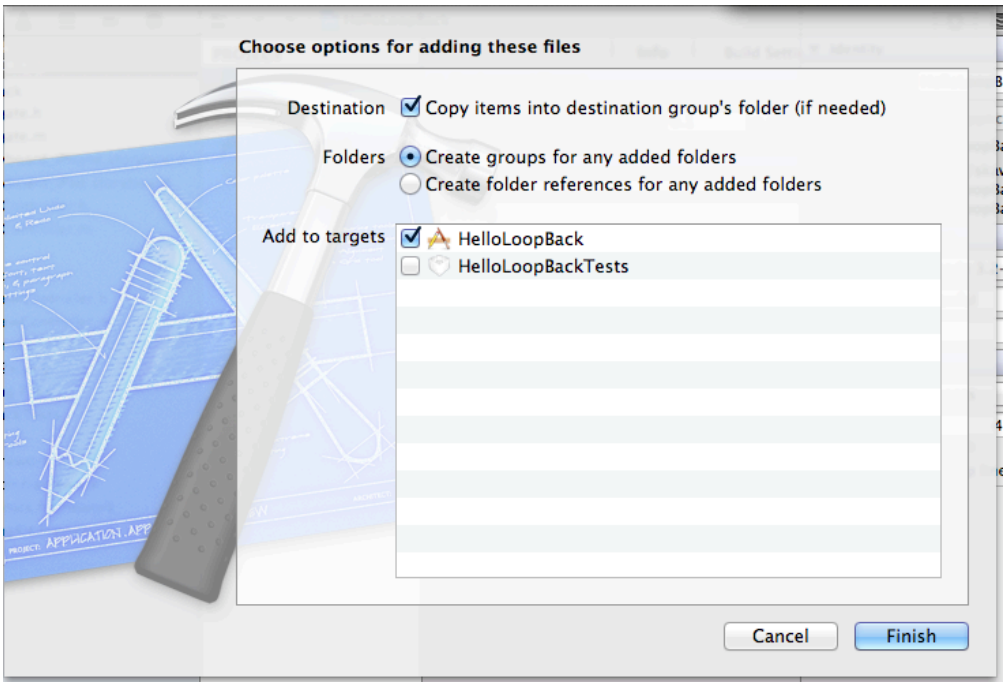
If you are creating a new iOS application or want to integrate an existing application with LoopBack, use the LoopBack SDK directly (LoopBack.framework), independent of the guide application.

Follow these steps:

1. Open the Xcode project you want to use with LoopBack, or create a new one.
2. Drag the entire LoopBack.framework folder from the new Finder window into your Xcode project.

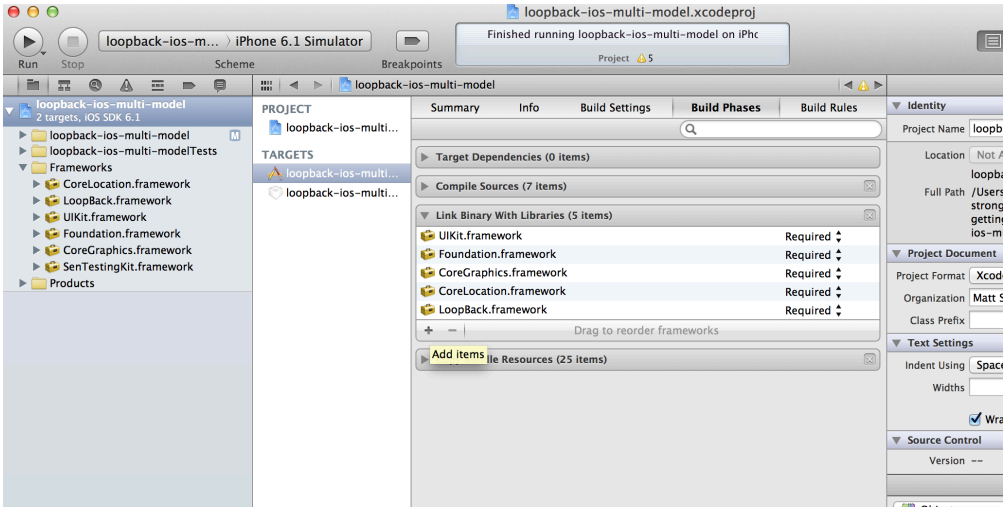


Important:Make sure to select "Copy items into destination group's folder". This places a copy of the SDK within your application's project folder.



3. Verify LoopBack is included in the list of iOS Frameworks to link against your binary. In your Project settings, check the 'Link with Binaries' section under the 'Build Phases' tab. If it's missing, add it directly by clicking the '+' button and selecting LoopBack.framework.

> ⚠ If **LoopBack.framework** isn't displayed in the list, try the previous step again; Xcode didn't create the copy it was supposed to create.



4. Import the LoopBack.h header into your application just as you would Foundation/Foundation.h. Type this line:

```
#import <LoopBack/LoopBack.h>
```

5. You need an `Adapter` to tell the SDK where to find the server. Enter this code:

```
LBRESTAdapter *adapter = [LBRESTAdapter adapterWithURL:[NSURL
URLWithString:@"http://example.com"]];
```

   This `LBRESTAdapter` provides the starting point for all our interactions with the running and anxiously waiting server.

   Once we have access to `adapter` (for the sake of example, we'll assume the Adapter is available through our AppDelegate), we can create basic `LBModel` and `LBModelRepository` objects. Assuming we've previously created a model named "product":

```
LBRESTAdapter *adapter = [[UIApplication sharedApplication] delegate].adapter;
LBModelRepository *productReposiory = [adapter
repositoryWithModelName:@"products"];
LBModel *pen = [Product modelWithDictionary:@{ "name": "Awesome Pen" }];
```

   All the normal `LBModel` and `LBModelRepository` methods (for example, `create`, `destroy`, and `findById`) are now available through `Product` and pen!
6. Go forth and develop! Check out the API docs or create more Models with the slc command-line tool.

Creating a sub-class of LBModel

Creating a subclass of LBModel enables you to get the benefits of an Objective-C class (for example, compile-time type checking).

Model interface and properties

As with any Objective-C class, the first step is to build your interface. If we leave any custom behavior for later, then it's just a few `@property` declarations and we're ready for the implementation.

```
/** * A widget for sale. */
@interface Widget : LBModel // This is a subclass, after all.

// Being for sale, each widget has a way to be identified and an amount of
// currency to be exchanged for it. Identifying the currency to be exchanged is
// left as an uninteresting exercise for any financial programmers reading this.
@property (nonatomic, copy) NSString *name;
@property (nonatomic) NSNumber *price;

@end
```

Model implementation

Since we've left custom behavior for later, just leave this here.

```
@implementation Widget
@end
```

Repository interface

The `LBModelRepository` is the LoopBack iOS SDK's placeholder for what in Node is a JavaScript prototype representing a specific "type" of Model on the server. In our example, this would be the model exposed as "widget" (or similar) on the server:

```
var Widget = app.model('widget', {
  dataSource: "db",
  properties: {
    name: String,
    price: Number
  }
});
```

Because of this the repository class name (`'widget'`, above) needs to match the name that model was given on the server.

If you haven't created a model yet, see Defining models. The model *must* exist (even if the schema is empty) before your app can interact with it.

Use this to make creating Models easier. Match the name or create your own.

Since `LBModelRepository` provides a basic implementation, we only need to override its constructor to provide the appropriate name.

```
@interface WidgetRepository : LBModelRepository

+ (instancetype)repository;

@end
```

Repository implementation

Remember to use the right name:

```
@implementation WidgetRepository

+ (instancetype)repository {
    return [self repositoryWithClassName:@"widget"];
}

@end
```

A little glue

Just as you did in Getting started, you'll need an `LBRESTAdapter` instance to connect to our server:

```
LBRESTAdapter *adapter = [LBRESTAdapter adapterWithURL:[NSURL
URLWithString:@"http://myserver:3000"]];
```

Remember: Replace "`http://myserver:3000`" with the complete URL to your server.

Once you have that adapter, you can create a repository instance.

```
WidgetRepository *repository = (WidgetRepository *)[adapter
repositoryWithModelClass:[WidgetRepository class]];
```

Using the repository instance

Now that you have a `WidgetRepository` instance, you can create, save, find, and delete widgets, as illustrated below.

Create a Widget:

```
Widget *pencil = (Widget *)[repository modelWithDictionary:@{ @"name": @"Pencil",
@"price": @1.50 }];
```

Save a Widget:

```
[pencil saveWithSuccess:^{
                    // Pencil now exists on the server!
                }
                failure:^(NSError *error) {
                    NSLog("An error occurred: %@", error);
                }];
```

Find another Widget:

```
[repository findWithId:@2
              success:^(LBModel *model) {
                  Widget *pen = (Widget *)model;
              }
              failure:^(NSError *error) {
                  NSLog("An error occurred: %@", error);
              }];
```

Remove a Widget:

```
[pencil destroyWithSuccess:^{
                        // No more pencil. Long live Pen!
                    }
                    failure:^(NSError *error) {
                        NSLog("An error occurred: %@", error);
                    }];
```

Creating a LoopBack iOS app: part one

- Download the server side code
- Download the iOS client application

This is part one of a two-part tutorial on creating a simple iOS app that connects to a LoopBack server application to perform create, read, update, and delete (CRUD) operations:

- Part one shows how to fetch data stored in the data source (in this case, the in-memory data source).
- Part two explains how to connect various interface elements to the app.

Prerequisites

Before starting this tutorial:

- **Install Apple XCode** version 5 or later.
- **Follow Getting started with LoopBack** to install the StrongLoop tools.
- **Download the LoopBack iOS SDK** and extract the contents.

Create the LoopBack books application

Follow these steps to create a simple LoopBack Node application using the StrongLoop command-line tool, slc:

1. Create a new application called books

```
$ slc loopback
[?] Enter a directory name where to create the project: books
[?] What's the name of your application? bookes
```

2. Follow the instructions in Create a LoopBack application to create a model called "book" with the default properties. Follow the instructions in the link to the section "Defining a model manually".
3. Run the app and load this URL in your browser to view the API explorer: http://localhost:3000/explorer.
4. Add a few books to the app by running a POST request through the API Explorer. The format of JSON for the POST request is as follows:

```
{"title" : "The Godfather",
"author" : "Mario Puzo",
"genre" : "Fiction",
"totalPages" : "1000",
"description" : "Classic novel of La Cosa Nostra"}
```

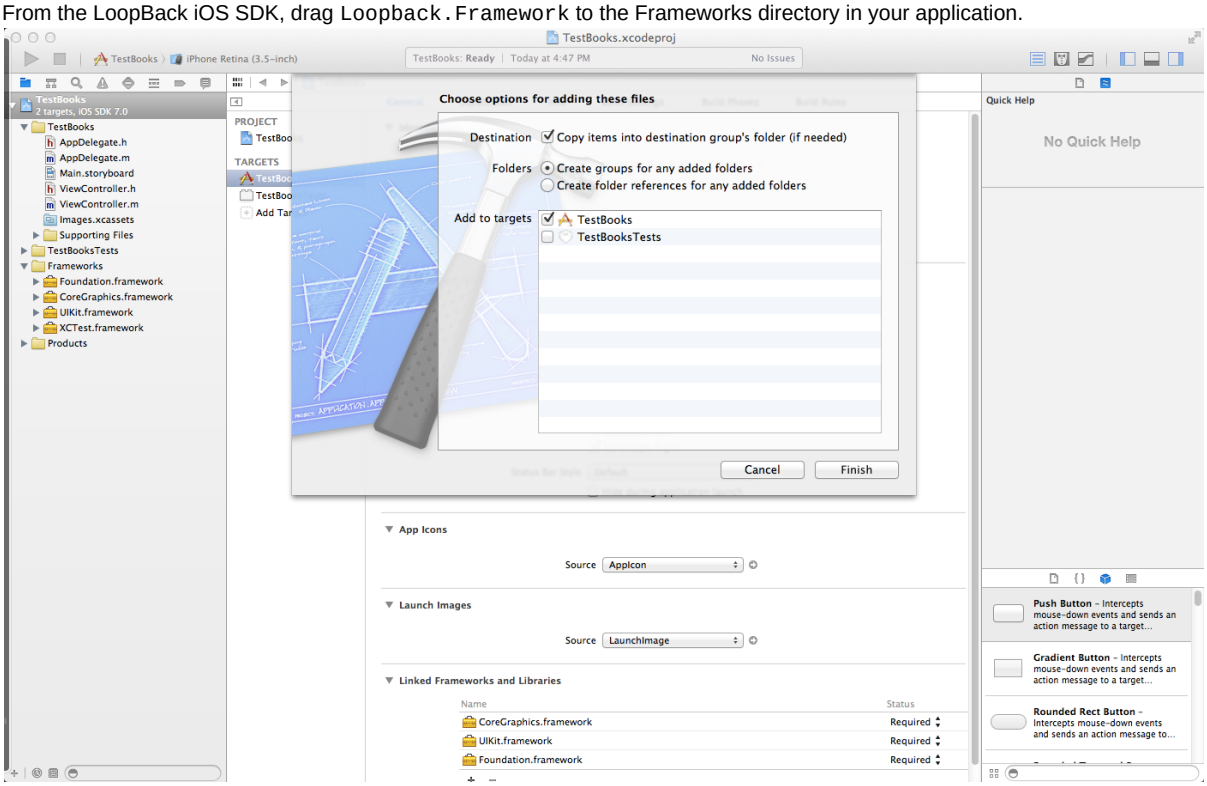
Create the iOS app

Follow these steps to create an iOS single view application:

1. For this example you . In Xcode, choose **File > New > Project > IOS Application > Single View Application**.
2. Name the project "Books" (or choose another name if you prefer). The instructions below assume the project is named "Books;" if you

use a different name, then some files will be named different accordingly.

3. Select **iPhone** as Device. Select location and create project. At this point, you should have a project created in Xcode with a bunch of default files.
4. Add Loopback to your application:
   a. From the LoopBack iOS SDK, drag `Loopback.Framework` to the Frameworks directory in your application.



   b. Import the Loopback framework into your app. Edit `booksAppDelegate.h` and add lines 2 and 7 as shown below:

booksAppDelegate.h

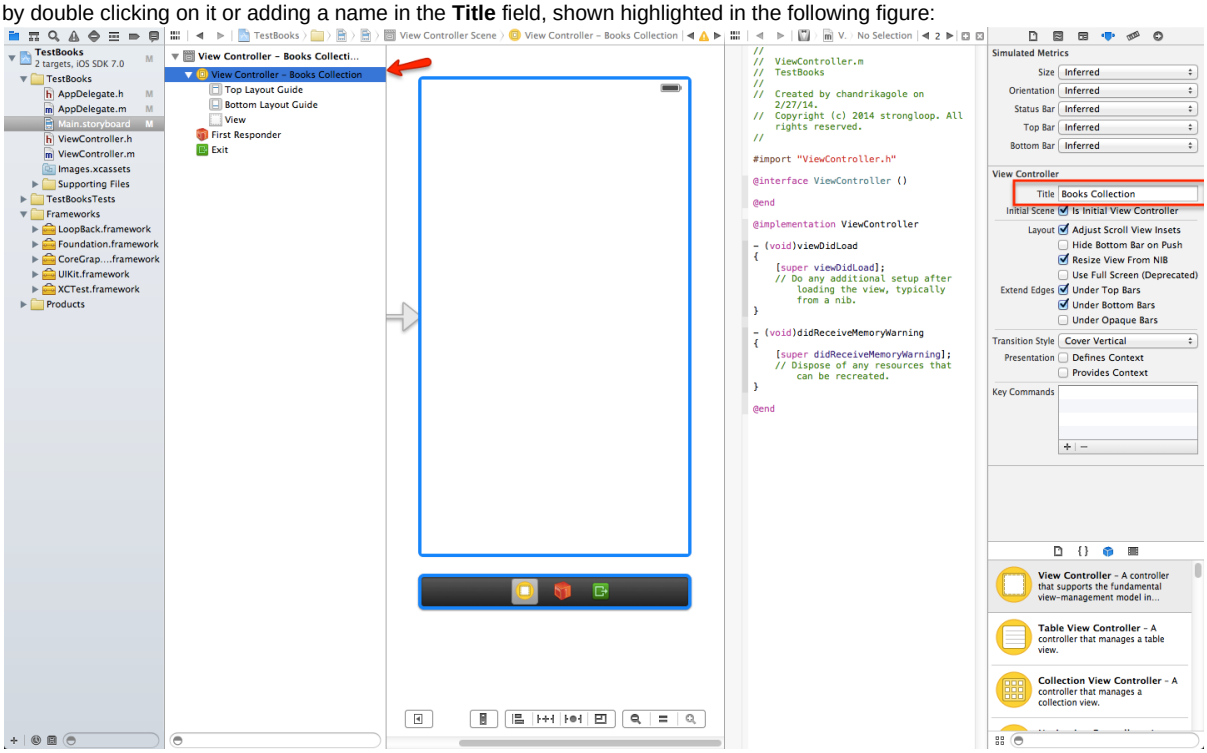
```
#import <UIKit/UIKit.h>
#import <LoopBack/LoopBack.h>

@interface booksAppDelegate : UIResponder <UIApplicationDelegate>

@property (strong, nonatomic) UIWindow *window;
+ ( LBRESTAdapter *) adapter;
@end
```

Edit `booksAppDelegate.m` to add the code in lines 3 through 11 as shown below:

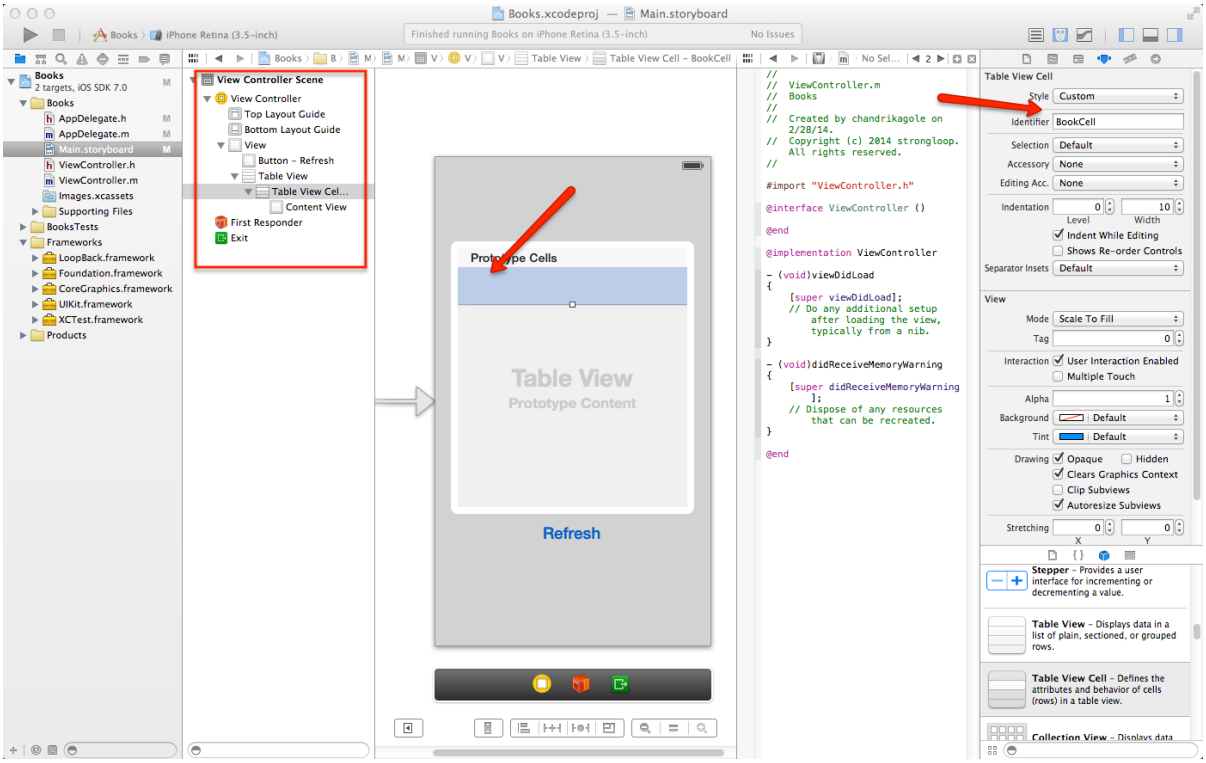
booksAppDelegate.m

```
#import "booksAppDelegate.h"
@implementation booksAppDelegate
static LBRESTAdapter * _adapter = nil;
+ (LBRESTAdapter *) adapter
{
    if ( !_adapter)
        _adapter = [LBRESTAdapter adapterWithURL:[NSURL
URLWithString:@"http://localhost:3000/api/"]];
    return _adapter;
}
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
...
```

5. Create the first screen for the application:
   a. Click on **Main.storyboard** in the File Navigator to see an empty View Controller. Name this View Controller "Books Collection" by double clicking on it or adding a name in the **Title** field, shown highlighted in the following figure:

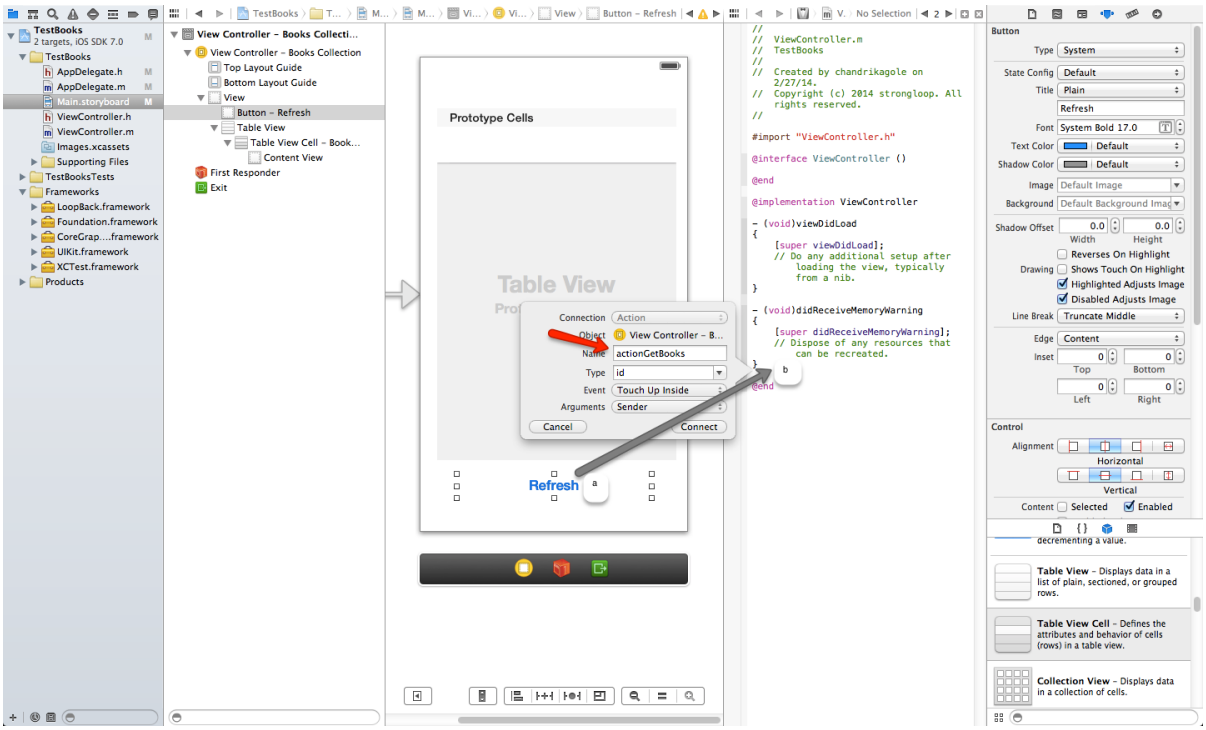


   b. **Design the Books Collection Screen.** Click on the box icon to list the Object Library items and drag a **Button** from the selection panel to the View Controller, near the bottom and centered horizontally. Change the **Title** field of the button from "Button" to "Refresh."
   c. Select and drag a **Table View** to the View Controller. Add a **Table View Cell** to the Table View. Click on the Table View Cell and enter "BookCell" as the **Identifier**. You will use this later. The View Controller should look like the screenshot below. Make sure the file hierarchy in the second panel is same as the screenshot.

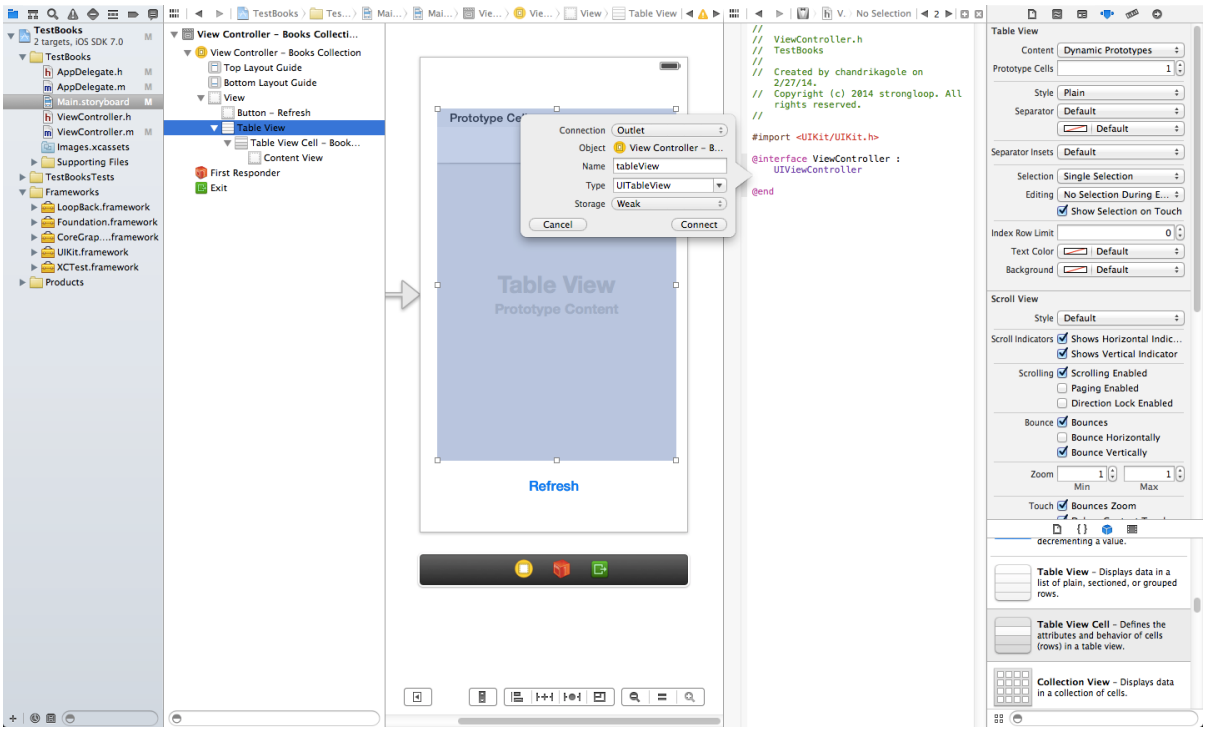


d. Connect the elements in the screen with the View Controller class: Select the Refresh button in your View Controller, hold Control and drag it into the `ViewController.m` right before `@end`. Name it "actionGetBooks" and click **Connect**, as shown below. This will insert code that looks like this:

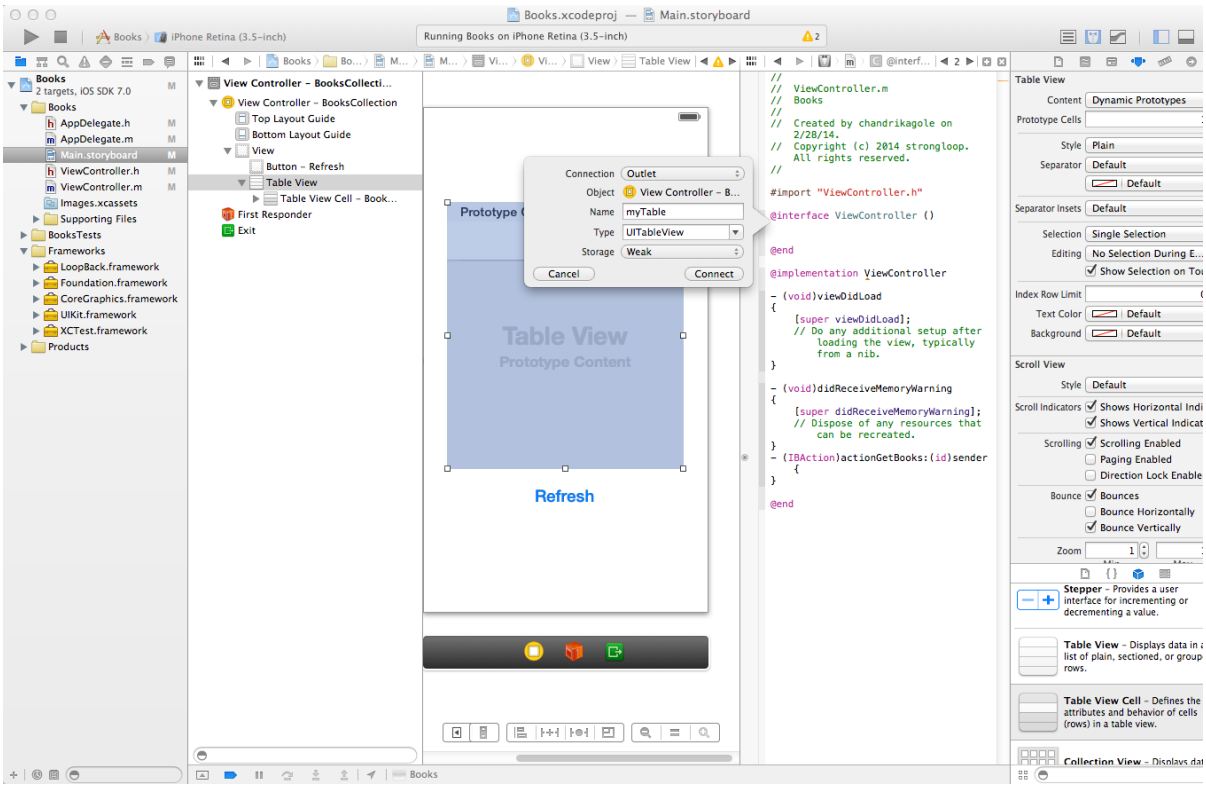
```
- (IBAction)actionGetBooks:(id)sender {
}
```



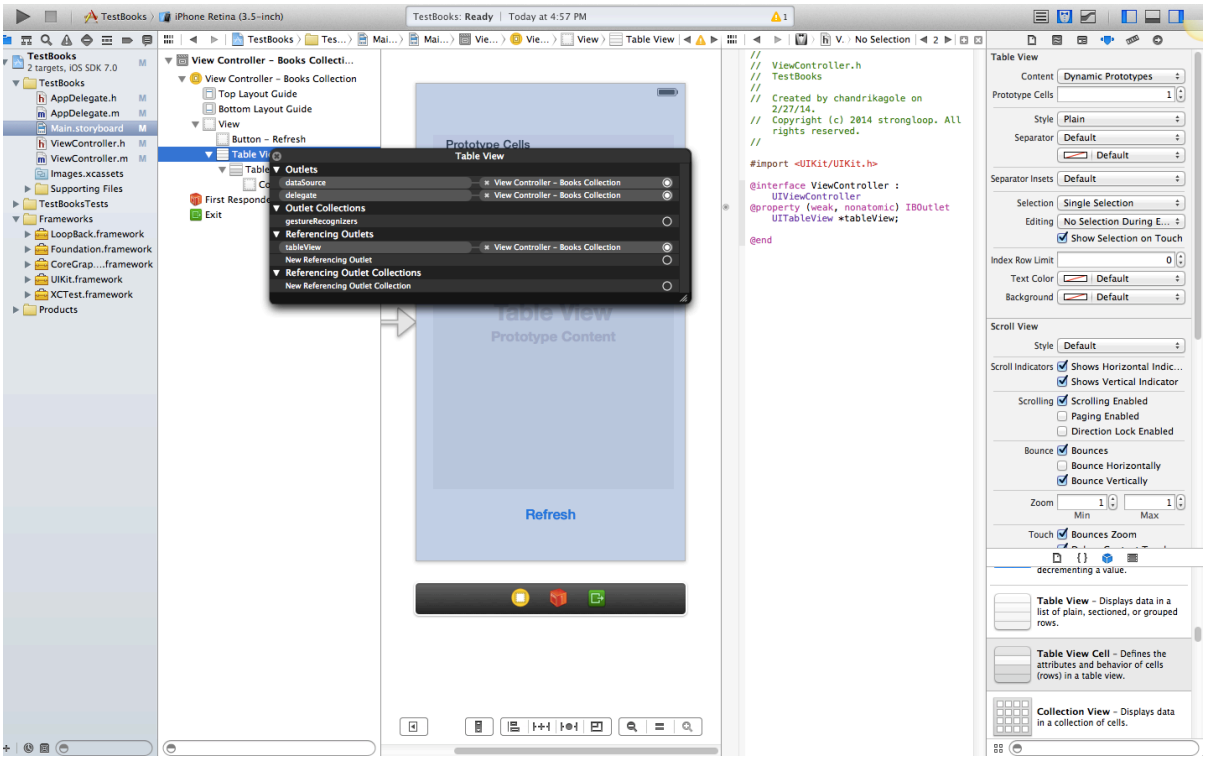
e. Define the TableView property by control-dragging into the `ViewController.h` file, as illustrated below.



f. Define `*myTable` by control-dragging into `ViewController.m`, as illustrated below.



g. Set outlets for the TableView by control-dragging them to the View Controller .
In the second pane, right-click on Table View. Under Outlets, click on dataSource and drag it to the Books View Controller. Click on delegate and also drag it to the Books View Controller.



h. Implement the 'actionGetBook' function.
   - Import `AppDelegate.h` in `ViewController.m`.
   - Define the interfaces for the table.
   - Define the `getBooks` function. Refer to the comments in the code below.

booksViewController.h

```
#import <UIKit/UIKit.h>
#import "booksAppDelegate.h"
@interface ViewController : UIViewController
@property (weak, nonatomic) IBOutlet UITableView *tableView;
@end
```

booksViewController.m

```
#import "booksViewController.h"
#import "booksAppDelegate.h"
#define prototypeName @"books"

@interface booksViewController ()
@property (weak, nonatomic) IBOutlet UITableView * myTable;
@property (strong, nonatomic) NSArray *tableData;
@end

@implementation booksViewController

...
- (NSArray *) tableData
{
```

```
    if (!_tableData)_tableData = [[NSArray alloc] init];
    return _tableData;
}
- (void) getBooks
{
    //Error Block
    void (^loadErrorBlock)(NSError *) = ^(NSError *error){
        NSLog(@"Error on load %@", error.description);
    };
    void (^loadSuccessBlock)(NSArray *) = ^(NSArray *models){
        NSLog(@"Success count %d", models.count);
        self.tableData = models;
        [self.myTable reloadData];
    };
    //This line gets the Loopback model "book" through the adapter
defined in AppDelegate
    LBModelRepository *allbooks = [[booksAppDelegate adapter]
repositoryWithModelName:prototypeName];
    //Logic - Get all books. If connection fails, load the error
block, if it passes, call the success block and pass allbooks to it.
    [allbooks allWithSuccess:loadSuccessBlock
failure:loadErrorBlock];
};

- (NSInteger) tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section
{
    return [self.tableData count];
}

// To display data in the table.
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *simpleTableIdentifier = @"BookCell";
    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:simpleTableIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc]
initWithStyle:UITableViewCellStyleDefault
reuseIdentifier:simpleTableIdentifier];
    }
    if (  [ [[self.tableData objectAtIndex:indexPath.row] class]
isSubclassOfClass:[LBModel class]])
    {
        LBModel *model = (LBModel *)[self.tableData
objectAtIndex:indexPath.row];
        cell.textLabel.text = [[NSString alloc] initWithFormat:@"%@ by
%@",
                               [model
objectForKeyedSubscript:@"title"], [model
objectForKeyedSubscript:@"author"]];
        //        cell.imageView.image = [UIImage
imageNamed:@"button.png"];
    }
```

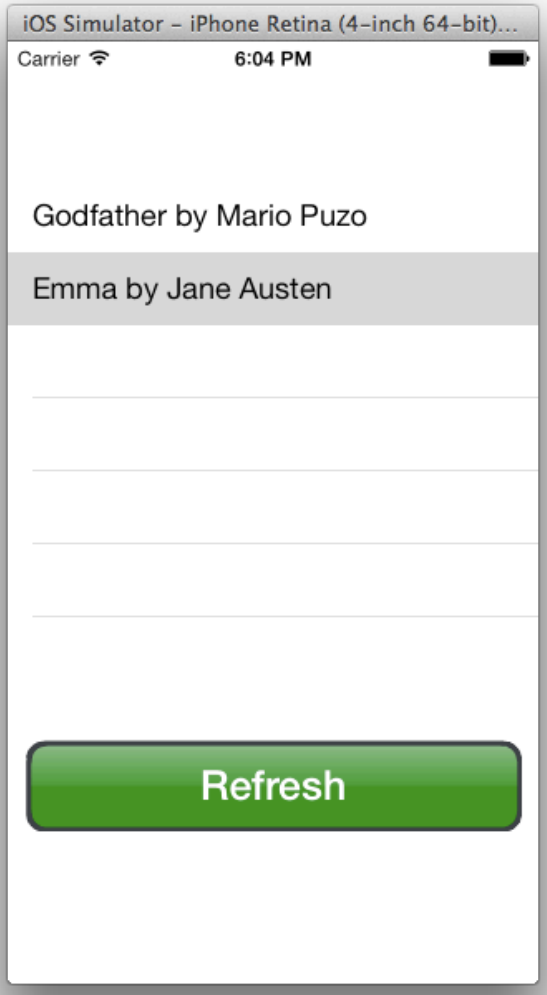
```
    return cell;
}
...
```

Call `getBooks` from the `actionGetBooks` function defined in step 4.c -

```
- (IBAction)actionGetBooks:(id)sender {
    [self getBooks];
}
```

Run the app in the simulator

At this point you should be able to build your app and get the list of books. Build the app and run it on a simulator. Click **Refresh** and you'll see the list of books.



Now continue to Part 2.

Creating a LoopBack iOS app: part two

- Download the server side code
- Download the iOS client application

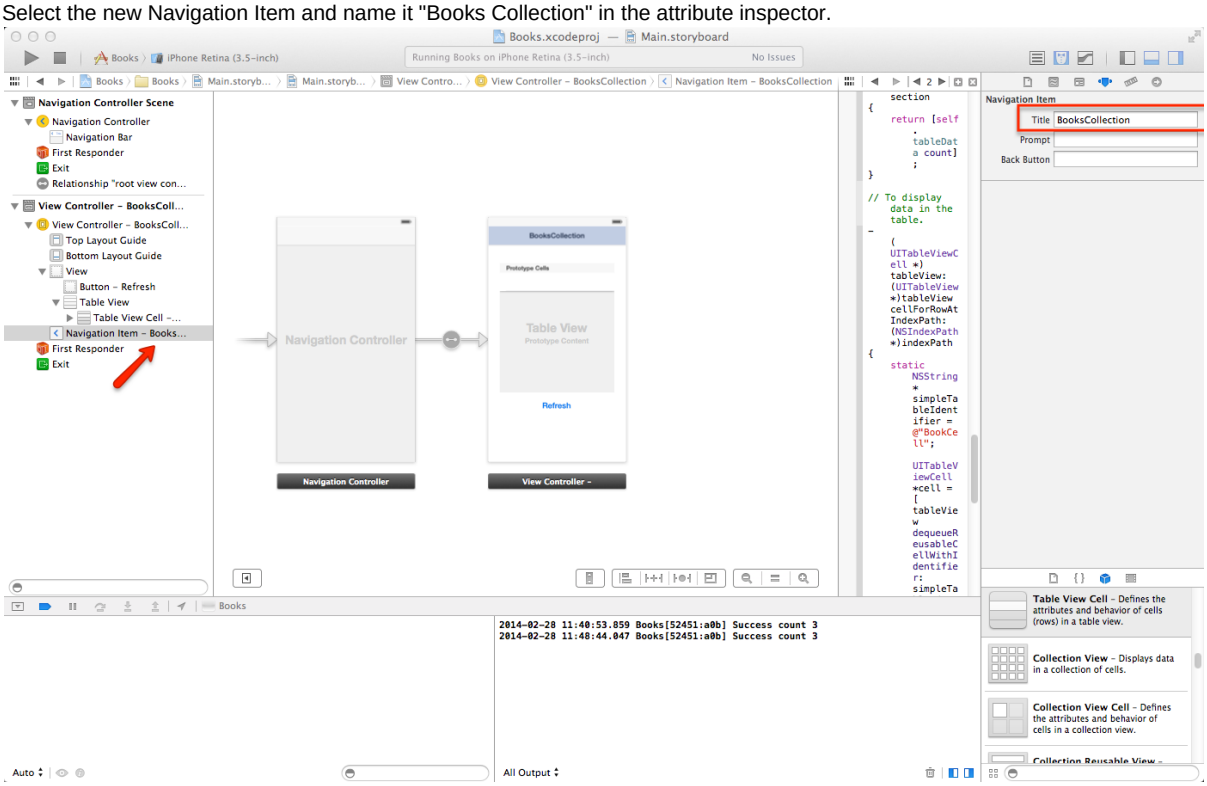
This is the second of a two-part tutorial on creating a simple iOS app that connects to a LoopBack server application to perform create, read, update, and delete (CRUD) operations.

If you haven't already done so, read part one before continuing.

Add navigation control

To add a navigation control to your app:

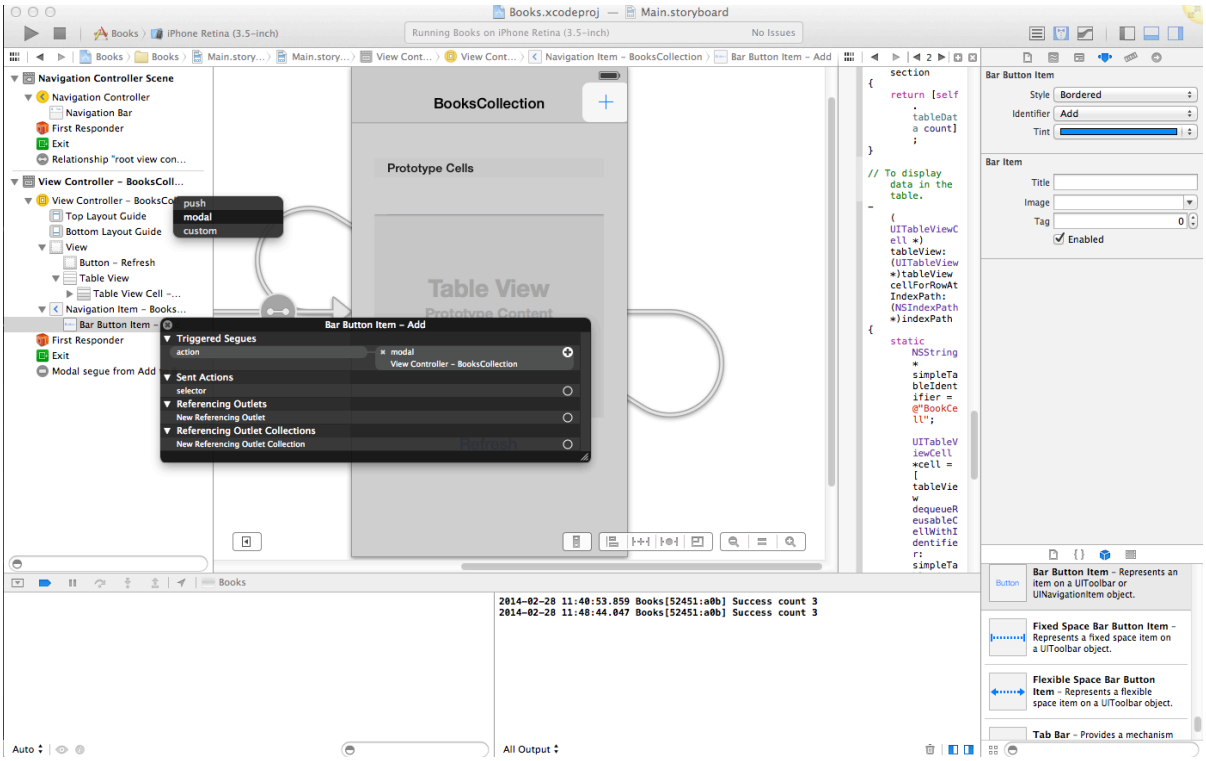
1. Select Books Collection View Controller.
2. Click **Editor > Embed In > Navigation Controller**. This will add a Navigation Item under the Books Collection View Controller, as shown below.
3. Select the new Navigation Item and name it "Books Collection" in the attribute inspector.



Implement the Add Book interface

To add the interface elements that enable a user to add a book:

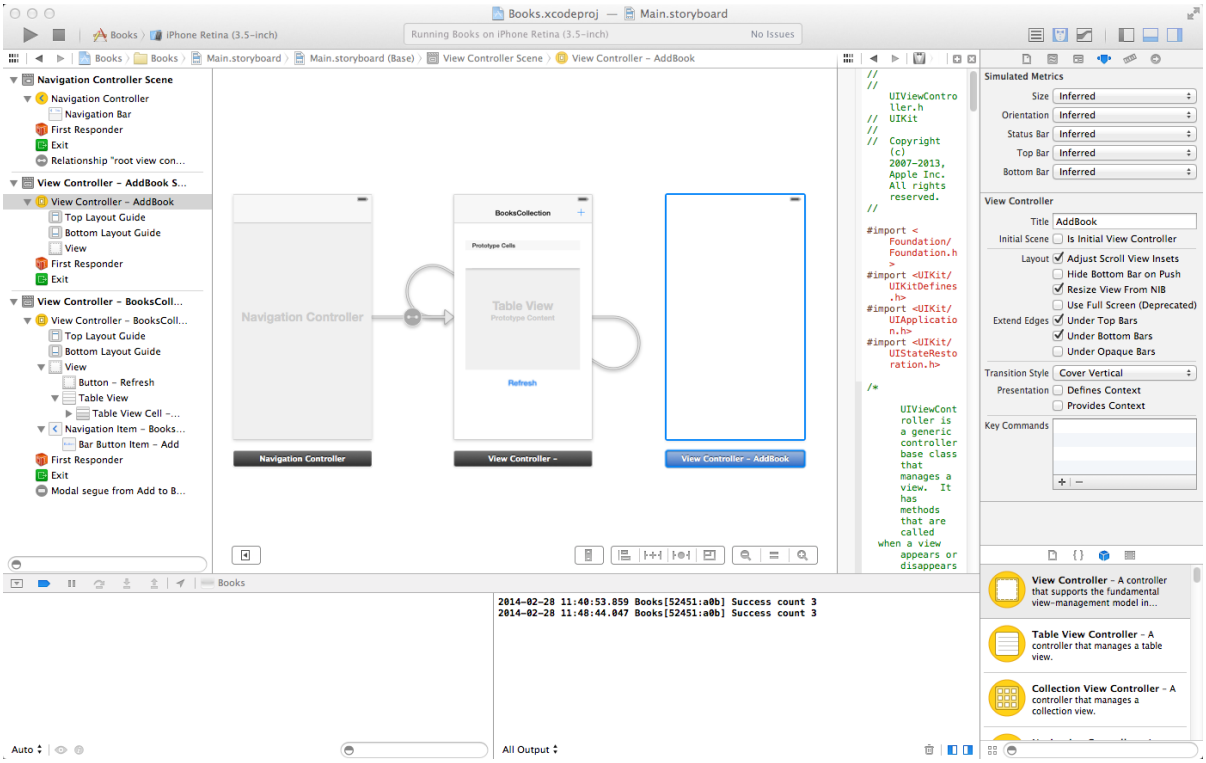
1. From the object library in lower right corner of the XCode window, select **Bar Button** Item.
2. Drag and drop it to the top right corner of the app shown in the XCode storyboard.
3. In the attribute inspector change **Identifier** to **Add**, as shown below.
4. Right click the Add bar button and Control-drag the circle next to action to the Books Collection View Controller. Select 'modal' as action.



Add another View Controller

When the user clicks the "+" button you want the app to show a screen to add a book to the collection. To do that you need to add another View Controller:

1. Drag a **View Controller** element from the Object Library into the storyboard.
2. Name this View Controller "Add Book".



3. Now connect the "+" button from the Books Collection View Controller to this screen: Control-dragg the "+" button from the **Books Collection View Controller** to the **Add Books View Controller**. This creates a segue.

4. Select **modal** as segue type.
5. Implement the segue action by adding the following code to `ViewController.m`:

ViewController.m

```
...
@implementation ViewController
//Add these 3 lines
- (IBAction)unwindToList:(UIStoryboardSegue *)segue
{
}
...
- (void)viewDidLoad
```

Add navigation to View Controller

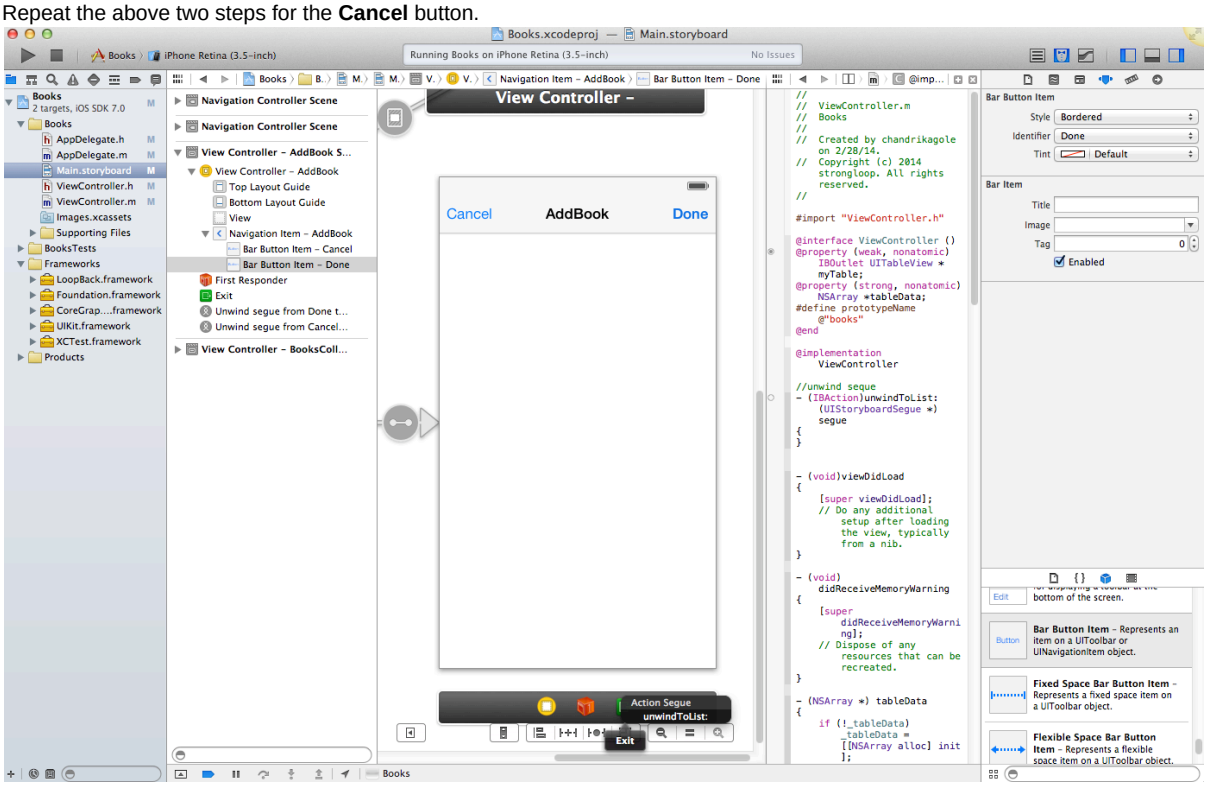
Now add navigation to the "Add Books" View Controller as follows:

1. Select the **Add Book View Controller**.
2. Choose **Editor > Embed In > Navigation Controller**.
3. Name this Navigation Controller "AddBook".

Add elements to Add Books sceen

Add a Done and Cancel button to the Add Books screen as follows:

1. In the **AddBook** scene, select **Navigation Item**.
2. In the Object Library, drag and drop two **Bar Button Items** to the scene.
3. Select each Bar Button Item and in the attribute inspector change their identifiers to "Done" and "Cancel".
4. Control-drag the **Done** button to the green exit button below the View Controller.
5. Select **unwindToList** as **Segue Action**.
6. Repeat the above two steps for the **Cancel** button.



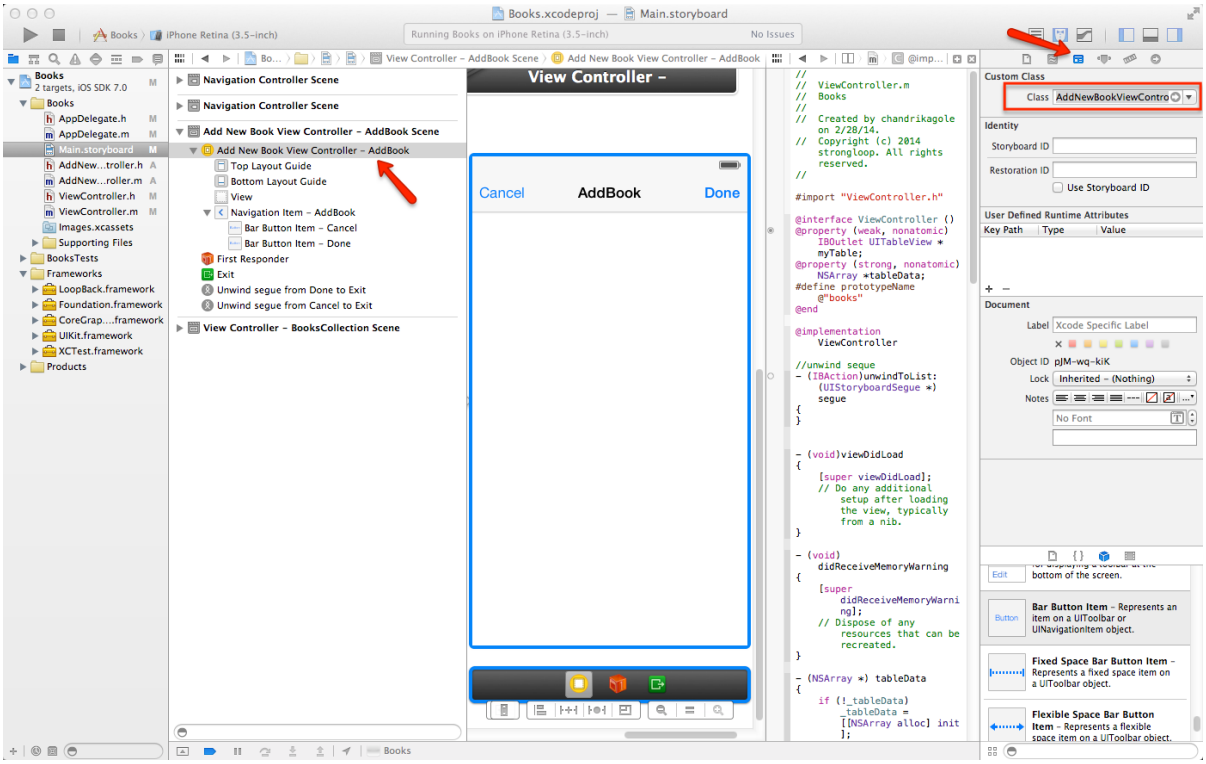
Add new class files

Add a new class for the Add Book ViewController as follows:

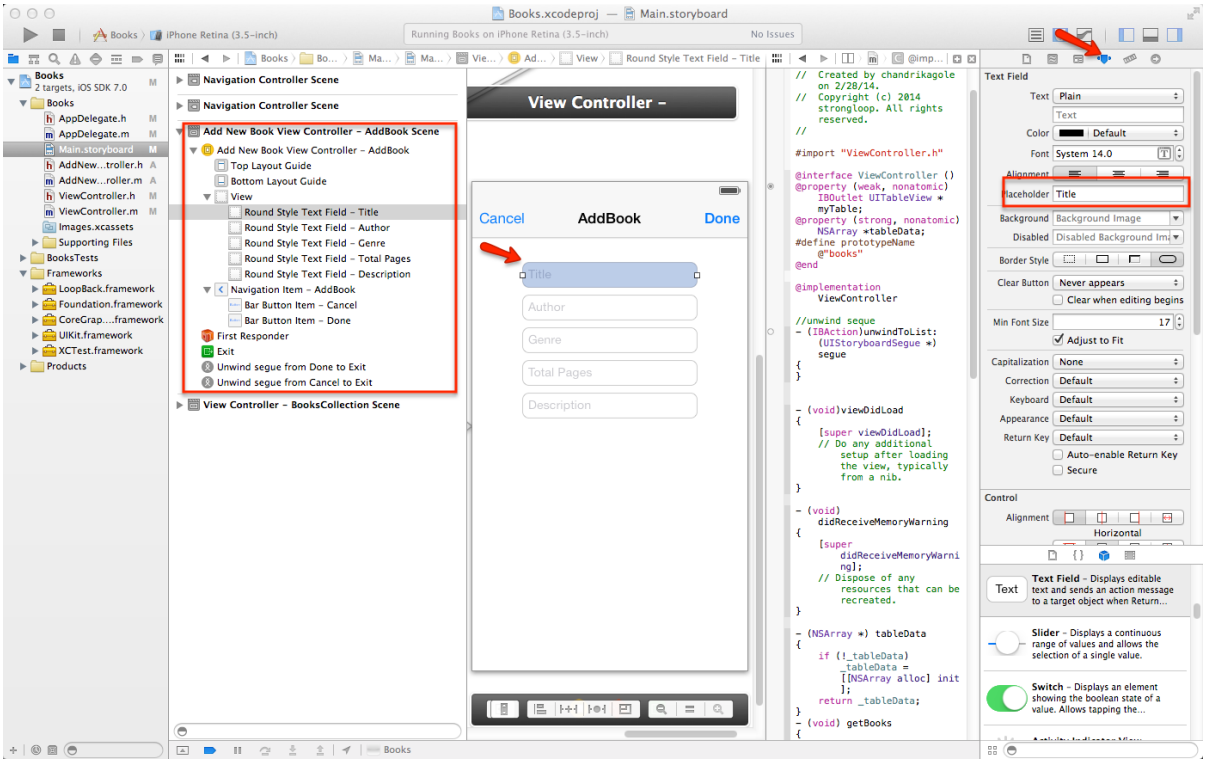
1. In the XCode menu, choose **File > New > File...**.
2. Select **Objective-C class** and click **Next**.
3. In the **Choose options for your new file** screen, change **Class name** to "AddNewBookViewController."
4. Click **Create** to add two new files to your project: `AddNewBookViewController.h` and `AddNewBookViewController.m`.

Connect class to View Controller

Connect the AddNewBookViewController class to the View Controller for Add New Book.



Add Text Fields from the Object Library to the Add View Screen. Add one Text Field for each of the five properties: title, author, description, totalPages and genre. Your screen should look like the screenshot below:

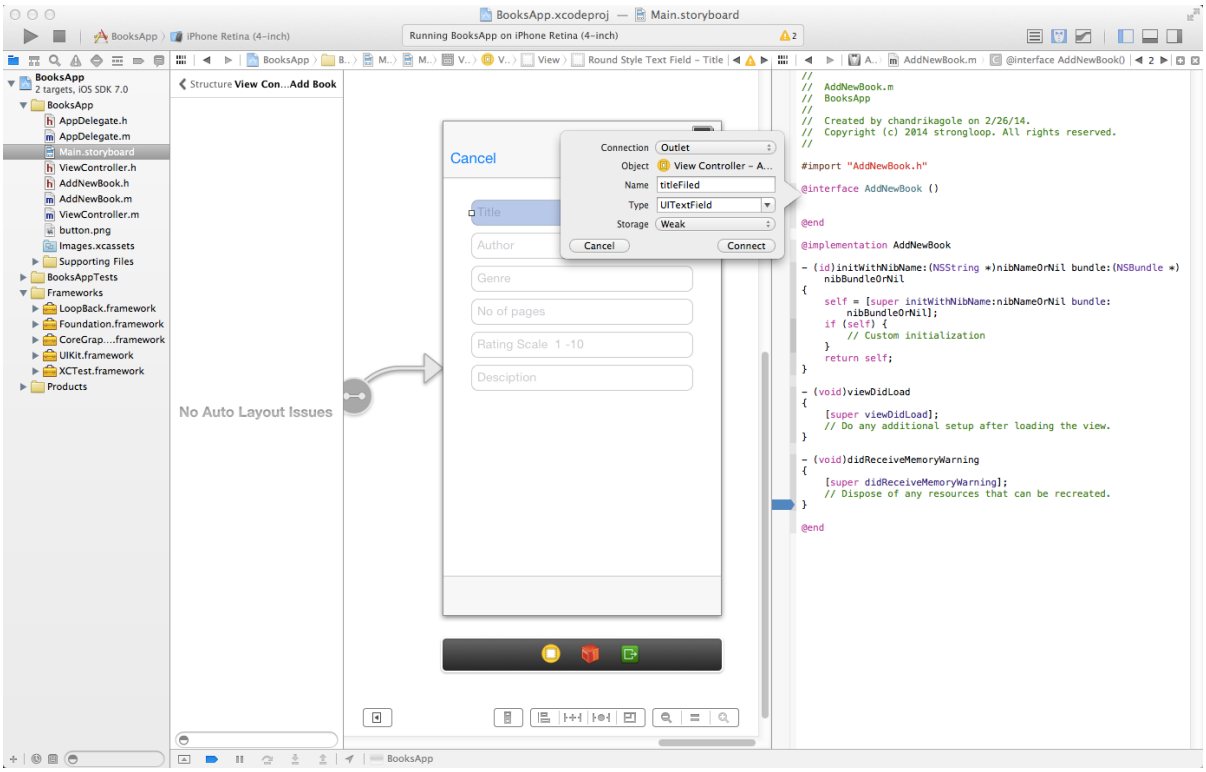


To connect the Text fields to your view controller:

1. Select the **Title** text field in your storyboard.
2. Control-drag from the text field on your canvas to the code displayed in the editor on the right, stopping at the line just below the `@interface` line in `AddNewBookViewController.m`.

In the dialog that appears, for Name, type "titleField."

Leave the rest of the options as they are. Your screen should look like this:

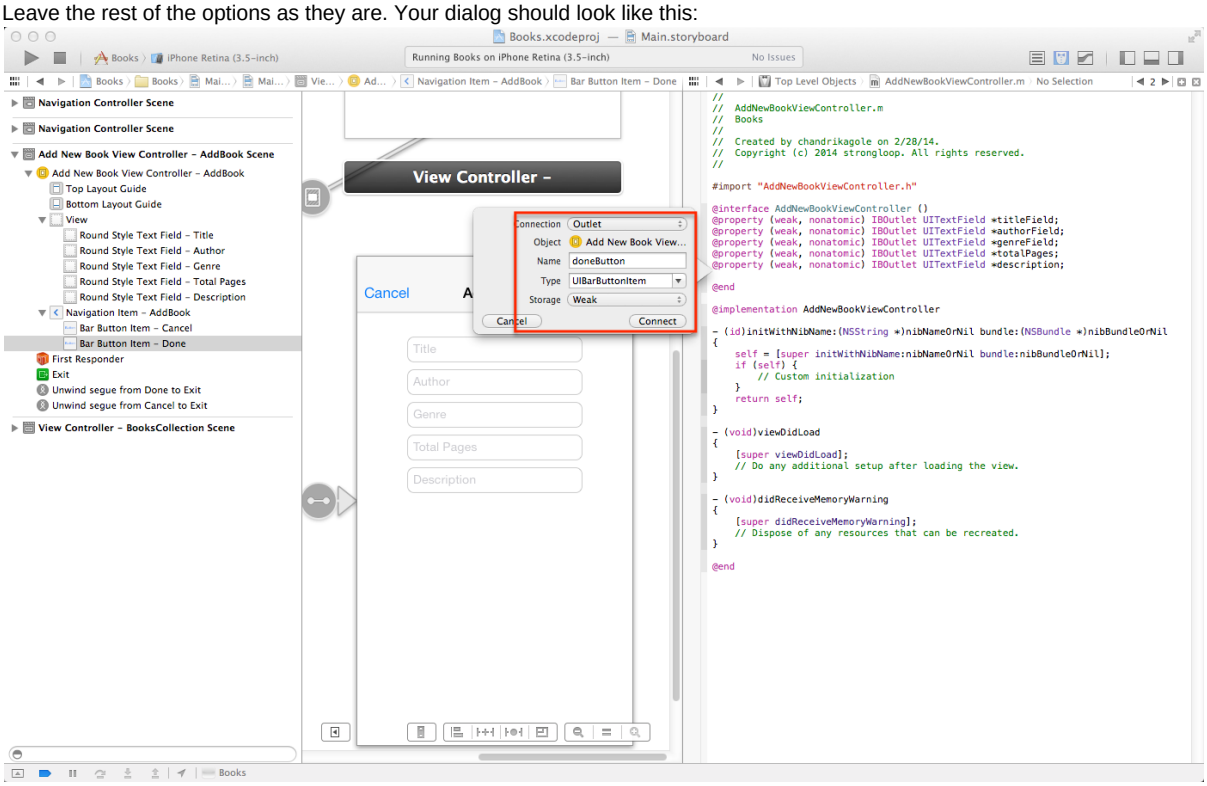


Do the same for the other text fields.

To connect the Done button to your view controller:

1. Select the Done button in your storyboard.
2. Control-drag from the Done button on your canvas to the code display in the editor on the right, stopping the drag at the line just below your `textField` properties in `AddNewBookViewController.m`.
3. In the dialog that appears, for Name, type "doneButton."

   Leave the rest of the options as they are. Your dialog should look like this:



Now add functionality to the class to save the book when the user adds one:

AddNewBookViewController.h

```
#import <UIKit/UIKit.h>
#import "ViewController.h"
...
```

AddNewBookViewController.m

```
#import "AddNewBook.h"
#import "ViewController.h"
#define prototypeName @"books"
@interface AddNewBook ()
@property (weak, nonatomic) IBOutlet UITextField *titleField;
@property (weak, nonatomic) IBOutlet UITextField *authorField;
@property (weak, nonatomic) IBOutlet UITextField *genreField;
@property (weak, nonatomic) IBOutlet UITextField *descriptionField;
@property (weak, nonatomic) IBOutlet UITextField *totalPagesField;
@property (weak, nonatomic) IBOutlet UIBarButtonItem *doneButton;
@end
@implementation AddNewBookViewController

- (void) prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    void (^saveNewErrorBlock)(NSError *) = ^(NSError *error){
        NSLog(@"Error on save %@", error.description);
    };
    void (^saveNewSuccessBlock)() = ^(){
        UIAlertView *messageAlert = [[UIAlertView alloc]initWithTitle:@"Successfull!"
message:@"You have add a new book" delegate:nil cancelButtonTitle:@"OK"
otherButtonTitles:nil];
        [messageAlert show];
    };
    if (sender != self.doneButton) return;
    if (self.titleField.text.length > 0) {
        NSLog(@"%@",self.titleField.text);
        LBModelRepository *newBook = [[AppDelegate adapter]
repositoryWithModelName:prototypeName];
        //create new LBModel of type
        LBModel *model = [newBook modelWithDictionary:@{
                                                        @"title"        :
self.titleField.text,
                                                        @"author"       :
self.authorField.text,
                                                        @"genre"        :
self.genreField.text,
                                                        @"totalPages"   :
self.totalPagesField.text,
                                                        @"description"  :
self.descriptionField.text
                                                        }];
        [model saveWithSuccess:saveNewSuccessBlock failure:saveNewErrorBlock];
    }
    else {
        UIAlertView *messageAlert = [[UIAlertView alloc]initWithTitle:@"Missing Book
Title!" message:@"You have to enter a book title." delegate:nil
cancelButtonTitle:@"OK" otherButtonTitles:nil];
        [messageAlert show];
    }
}
....(ctd)
```

Run the build and try it out. You should be able to add a new book and refresh to fetch all books.

Using Cocoapods in Swift with iOS SDK

Overview

CocoaPods manages library dependencies for your Xcode projects using a single text file called a Podfile.

Install Cocoapods

You can skip this step if you already have cocoapods installed on your Mac. Otherwise simply open your terminal and run the following command:

```
sudo gem install cocoapods
```

That's it, you're now ready to use Cocoapods!

Create a podfile

Close your XCode project and in the main folder create a new file named "Podfile".

Insert the following:

```
pod 'LoopBack', :git => 'https://github.com/strongloop/loopback-sdk-ios.git'
```

Install the SDK

Install the Cocoapods running the following command in your project directory:

```
$ pod install
```

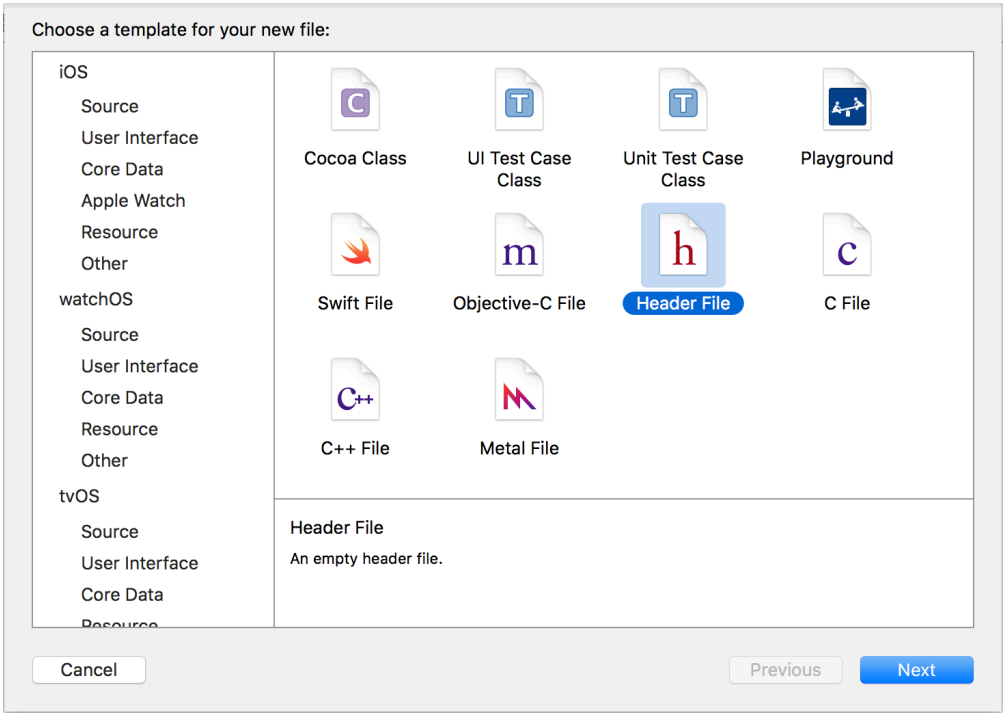
Cocoapods will now look for the podspec file in the GitHub repo and download all necessary frameworks.

Create the Swift bridge

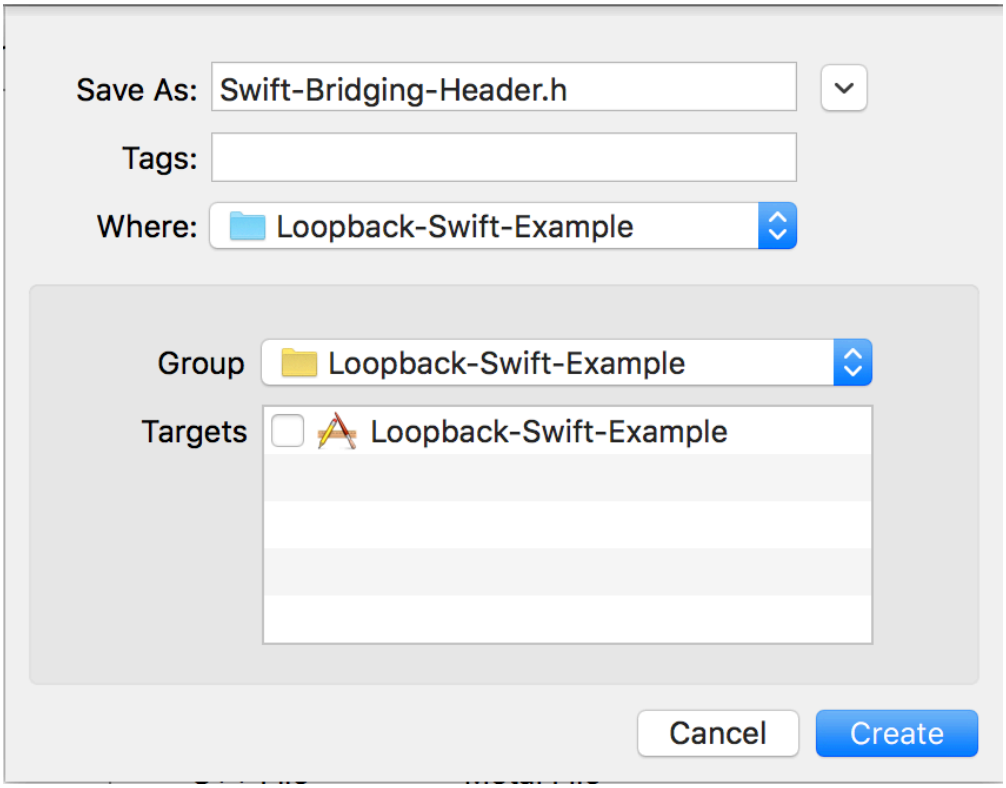
From now on you'll use the xcworkspace file in your project directory to open your app.

So go ahead and open the xcworkspace file to fire up XCode.

Select **File > New > File** and select **Header File**.



Enter `Swift-Bridging-Header.h` as the file name.



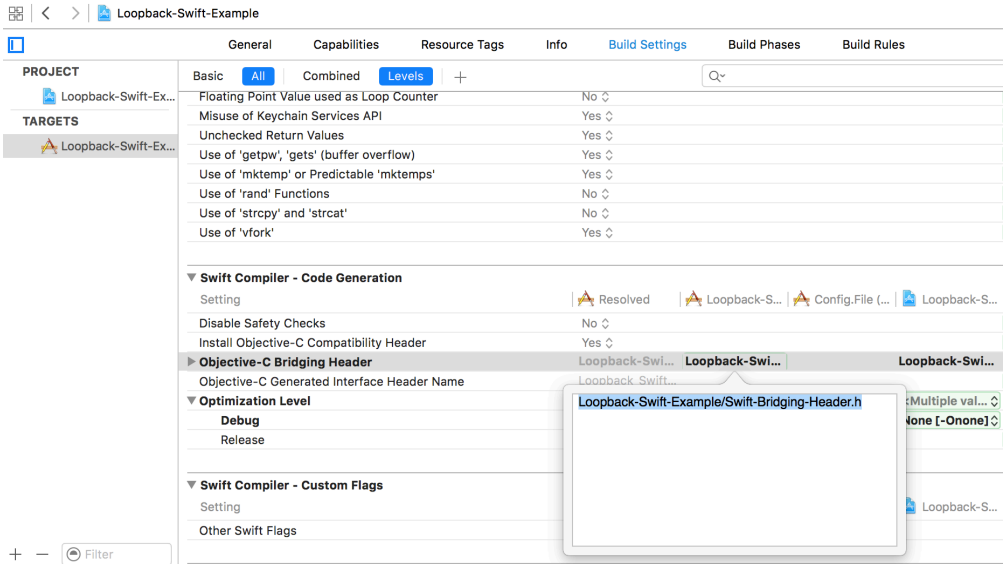
Now open the file and simply insert the following line:

```
#import <LoopBack/LoopBack.h>
```

Afterwards all you need to do is include the bridging file into the Build Settings.

Open your projects Build Settings and look for **Objective-C Bridging Header**.

led to insert the path to the file you just created.



You're now ready to use the SDK in your Swift project.

AngularJS JavaScript SDK

Related articles:

See also: Angular SDK built-in models API

The LoopBack AngularJS SDK is automatically installed when you follow the installation instructions in Getting started with LoopBack.

The AngularJS SDK requires Angular version 1.2.0 or newer.

Introduction

AngularJS is an open-source JavaScript model–view–controller (MVC) framework for browser-based applications.

The LoopBack AngularJS SDK has three major components:

- Auto-generated AngularJS services, compatible with `ngResource.$resource`, that provide client-side representation of the models and remote methods in the LoopBack server application.
- Command-line tools:
  - `lb-ng` that generates Angular $resource services for your LoopBack application.
- A Grunt plugin (grunt-loopback-sdk-angular), if you want to use Grunt instead of `lb-ng`.

The client is dynamic, in other words it automatically includes all the LoopBack models and methods you've defined. You don't have to manually write any static code.

The SDK fits seamlessly into the workflow of a front-end developer:

- The generated Angular objects and methods have ngdoc comments. Use an ngdoc viewer like Docular to view documentation of the client available to your AngularJS client.
- If you wish, you can use the provided Grunt task to generate the client services script, which make it easy to include this file in an existing Grunt-based workflow (for example for bundling or "minification").

How it works

The code generator (either the `lb-ng` tool or a Grunt task) loads your LoopBack server application, walks through all models, and generates code registering an AngularJS factory for each model. The factory creates an `ngResource.$resource` object, passing in a description of all public methods the model class exposes. This way the resource object provides an API very similar to that of your backend Model class.

The rest of the procedure is standard for AngularJS: configure your client app to include and load the `lbServices` module, and tell Angular's injector which models to use in your code.

At runtime, the generated lbServices module (`lb-services.js`) depends on the Angular resource script (`angular-resource.js`) so you must ensure that your client loads this Angular script.

Generating Angular services

Use the LoopBack Angular command-line tool, `lb-ng`, to generate the Angular client library for your LoopBack application.

Any time you modify or add models to your LoopBack app, you must re-run `lb-ng` to re-generate the Angular client library to reflect the changes.

For example, if your application has the standard LoopBack project layout, then in the `/client` sub-directory, enter these commands:

```
$ mkdir js
$ lb-ng ../server/server.js js/lb-services.js
```

In this example:

`../server/server.js` is the relative path to the main LoopBack server script.

`js/lb-services.js` is the name and path to the JavaScript file that the tool generates.

If you want the AngularJS files to be somewhere else at runtime, then after generating the JavaScript file as shown above, simply copy it to the desired location and reference that location in your `index.html` file or equivalent.

The SDK also provides a tool to generate API documentation for the AngularJS services; see Generating Angular API docs for more information.

lb-ng command

The general syntax of the lb-ng command is:

lb-ng [*options*] *path-to-server-script* [*path-to-generated-services*]

ARGUMENTS:

`path-to-server-script`
Relative path to the LoopBack application main script. In the standard project layout, *<app-dir>*`/server/server.js`. Required.

`path-to-generated-services`
Relative path to the client JavaScript file to be generated containing the AngularJS $resource services. Optional; default is stdout.

OPTIONS:

`-m, --module-name [name]`
The name for generated Angular module. Default is "lbServices".

`-u, --url [url]`
URL of the REST API endpoint.

Using the SDK

The SDK creates $resource services that you can use in the same way as you would use hand-written ones. See the example in the AngularJS $resource docs.

There is one service for each model class, with the same name as the model name (for example, `User`). On top of the standard get/save/query/remove methods, the service includes functions for calling every method exposed by the remote model (for example, `User.login()`).

⚠ AngularJS model names start always with a capital letter, even if your server definition starts with a lower-case letter.

Setup

Follow these steps to use the generated services inside your AngularJS application:

1. Include `js/lb-services.js` in your `index.html` file

```
<script src="js/lb-services.js"></script>
```

ⓘ You'll need to set up static middleware to serve the client script. See Defining middleware for more information.

2. Register the AngularJS module `lbServices` as a dependency of your app. Add `'lbServices'` to the `angular.module()` call in the main JavaScript file of your Angular (client) app, as follows:

```
angular.module('my-app-module',
  ['ngRoute' /* etc */, 'lbServices', 'my-app.controllers'])
```

3. To call a method on a model from your controller, add the model name as a dependency of the controller; for example:

```
 // access User model
module.controller('LoginCtrl', function($scope, User, $location) {
  $scope.login = function() {
    $scope.loginResult = User.login($scope.credentials,
      function() {
        // success
      }, function(res) {
        // error
      });
```

Client configuration

You can configure some aspects of the generated services within the AngularJS client application using the LoopBackResourceProvider object, as illustrated below. This object is available to configuration blocks only; for more information, see Module Loading & Dependencies in the AngularJS documentation.

app.js

```
angular.module('my-app-module')
  .config(function(LoopBackResourceProvider) {

    // Use a custom auth header instead of the default 'Authorization'
    LoopBackResourceProvider.setAuthHeader('X-Access-Token');

    // Change the URL where to access the LoopBack REST API server
    LoopBackResourceProvider.setUrlBase('http://api.example.com/');
  });
```

See LoopBackResourceProvider API docs for the list of all available options.

Model resource API

Angular's `$resource` objects have a different API compared to LoopBack's API on the server:

- Method arguments are passed in an object (a key-value map), not positionally.
- Methods accept two optional callbacks (a success callback and an error callback).
- All methods return a special object populated with the response data once it arrives.
- The returned object has a special property `$promise` that you can use for promise-based flow control.

See "Returns" section in $resource Usage docs for more information about the AngularJS API and AngularJS client API docs for a reference of methods available on the built-in loopback models.

Example

Look at the method `Product.find(filter, callback)` to illustrate the differences.

On the server, one can write the following code to find all products named 'Pen':

server-code.js

```
Product.find(
  {
    where: { name: 'Pen' }
  },
  function(err, list) { /*...*/ });
```

The AngularJS code to perform the same operation is:

angular-code.js

```
$scope.products = Product.find(
  { filter: { where: { name: 'Pen' } } },
  function(list) { /* success */ },
  function(errorResponse) { /* error */ }
);
```

CRUD operations

The API for manipulating model instances is similar to the server-side Node API: it provides methods for standard create, read, update, and delete (CRUD) operations.

Create

The create method takes an object containing a key-value map of values for the model.

```
$scope.product = Product.create({ name: 'Pen' });
```

Read (querying)

You can fetch model data using the usual methods like `find()`, `findOne()` or `findById()`. Just don't forget to wrap the arguments in a key/value parameters map.

```
$scope.products = Product.find({
  filter: { limit: 10 }
});

$scope.pen = Product.findOne({
  filter: { where: { name: 'Pen' } }
});

$scope.prod0 = Product.findById({ id: 0 });
```

Refer to Querying data for a description of all query options. The AngularJS client expects the "Node syntax" of arguments, with the differences noted above.

Also, see Querying related models.

Update

After you obtain an instance of a Model resource using one of the query methods described above, persist changes back to the server by calling `$save` method:

```
$scope.product.price = 13.40;
$scope.product.$save();
```

Note that the method returns a promise and it also updates the model instance using the data returned by the server.

You can also perform a (partial) update with `prototype$updateAttributes` method:

```
$scope.product.price = 13.40;
Product.prototype$updateAttributes(
   { id: $scope.product.id },
   { price: $scope.product.price });
```

Delete

Use `deleteById()` to remove a model instance with a given ID.

```
Product.deleteById({ id: $scope.product.id })
  .$promise
  .then(function() { console.log('deleted'); });
```

Working with related models

The AngularJS SDK provides convenient methods for accessing and manipulating related models.

Currently, the AngularJS SDK does not support polymorphic belongsTo relations; an application with such a relation will display a warning such as this (for example):

```
Warning: scope Picture.imageable is missing _targetClass property.
The Angular code for this scope won't be generated. Please upgrade
to the latest version of loopback-datasource-juggler to fix the problem.
```

For example, consider the two following model definitions:

common/models/products.json

```
{
  "name": "Product",
  "properties": { /* ... */ },
  "relations": {
    "category": {
      "type": "belongsTo",
      "model": "Category"
    }
  }
}
```

common/models/categories.json

```
{
  "name": "Category",
  "properties": { /* ... */ },
  "relations": {
    "products": {
      "type": "hasMany",
      "model": "Product"
    }
  }
}
```

Querying related models

The SDK creates a `Category.products()` method that can be used to list all products in a given category.

```
$scope.products = Category.products({
  id: $scope.category.id,
  filter: {
    where: { name: 'Pen' },
    limit: 10
  }
});
```

Creating a related model

To create a new product in a given category, use `Category.products.create()`:

```
$scope.newProduct = Category.products.create(
  { id: $scope.category.id },
  { name: 'Pen' });
```

Removing related models

Last but not least, there is also a method to delete all related models (all products in a given category):

```
Category.products.destroyAll({ id: $scope.category.id });
```

Authentication

The SDK provides first-class support for authenticated requests.

Basic use is simple: Call `User.login()` with correct credentials and the SDK will handle everything else for you. It will store the access token provided in login response and send this token in all subsequent requests.

```
User.login({
  email: 'name@example.com',
  password: 'passwd'
});
```

Other useful methods include `User.isAuthenticated()`, to determine if the current user is logged in, and `User.getCurrentId()`, to get the ID of the currently logged-in user.

Persisting the access token

By default, the AngularJS SDK stores the access token in the browser's localStorage, so it is preserved across page reloads and browser (hybrid mobile app) restarts.

While this behavior is useful for hybrid mobile applications, traditional web applications typically need to store the token only for the duration of a browsing session, unless the user checked a "remember me" option. Fortunately this is easy to implement too: Just add `rememberMe` parameter to your `User.login()` call.

```
User.login({ rememberMe: $scope.rememberMe }, credentials);
```

Handling 401 Unauthorized

When there is no user logged in or the access token expires, requests to the LoopBack server fail with 401 errors. The application should install a global handler that will allow the user to login and repeat the action.

AngularJS provides HTTP interceptors that can be used to implement this feature.

```
// Inside app config block
$httpProvider.interceptors.push(function($q, $location, LoopBackAuth) {
  return {
    responseError: function(rejection) {
      if (rejection.status == 401) {
  //Now clearing the loopback values from client browser for safe logout...
        LoopBackAuth.clearUser();
        LoopBackAuth.clearStorage();
        $location.nextAfterLogin = $location.path();
        $location.path('/login');
      }
      return $q.reject(rejection);
    }
  };
});

// In the Login controller
User.login($scope.credentials, function() {
  var next = $location.nextAfterLogin || '/';
  $location.nextAfterLogin = null;
  $location.path(next);
});
```

The `responseError` interceptor saves the current location (line 6) and redirects to the login page (line 7). The login controller then redirects back to the previous page (lines 16-18) when the login was successful.

Angular example app

This article is reproduced from loopback-example-angular

loopback-example-angular

```
$ git clone https://github.com/strongloop/loopback-example-angular.git
$ cd loopback-example-angular
$ npm install
$ node . # then browse to localhost:3000
```

A simple todo list using AngularJS on the client-side and LoopBack on the server-side.

Prerequisites

Tutorials

- Getting started with LoopBack
- Tutorial series, step 1

Knowledge of

- Angular
- Angular Resource
- AngularUI Router
- Bootstrap
- Bower
- LoopBack
- LoopBack AngularJS SDK
- LoopBack models

- LoopBack middleware

Procedure

Create the application

Application information

- Name: `loopback-example-angular`
- Directory to contain the project: `loopback-example-angular`

```
$ slc loopback loopback-example-angular
... # follow the prompts
$ cd loopback-example-angular
```

Create the Todo model

Model information

- Name: `Todo`
- Data source: `db (memory)`
- Base class: `PersistedModel`
- Expose over REST: `Yes`
- Custom plural form: *Leave blank*
- Properties:
  - `content`
  - String
  - Required

```
$ slc loopback:model Todo
... # follow the prompts
```

Configure the vendor directory

In the project root, create `.bowerrc`.

> *Bower installs packages in `bower_components` by default, but we reconfigure this to `client/vendor` to make importing files into `index.html` more convenient.*

Install client-side dependencies

From the project root, run:

```
$ bower install angular angular-resource angular-ui-router bootstrap
```

Create the home page

Create `index.html` in the `client` directory.

Create the main stylesheet

Create a `css` directory to store stylesheets.

```
$ mkdir client/css
```

In this directory, create `styles.css`.

Serve static assets from the `client` directory

Delete `/server/boot/root.js`.

Add static middleware to the `files` section in `middleware.json`
.

Create `app.js`

Create a `js` directory to hold scripts files.

```
$ mkdir client/js
```

In this directory, create `app.js`.

> *Notice we declare `lbServices` as a dependency. We will generate this file using the `lb-ng` command in a later step.*

Create `todo.html`

Create a `views` to store view templates.

```
$ mkdir client/views
```

In this directory, create `todo.html`.

Create `controllers.js`

Create a `controllers` directory to store controller files.

```
$ mkdir client/js/controllers
```

In this directory, create `todo.js`.

Generate `lb-services.js`

Create a `services` directory to store service files.

```
$ mkdir client/js/services
```

`lb-ng` is automatically installed along with the `slc` command-line tool (ie. when you ran `npm install -g strongloop`). `lb-ng` takes two arguments: the path to `server.js` and the path to the generated services file.
`lb-services.js` is an Angular service used to interact with LoopBack models.

Create `lb-services.js` using the `lb-ng` command.

```
$ lb-ng server/server.js client/js/services/lb-services.js
```

Run the application

From the project root, enter `node .` and browse to `localhost:3000` to view the application.

Other LoopBack examples

AngularJS Grunt plugin

- Overview

Overview

If you prefer, you can use a Grunt plugin to auto-generate Angular $resource services instead of the LoopBack Angular tool, `lb-ng`. You can then use grunt-docular.com to generate client API docs.

Installation

```
$ npm install grunt-loopback-sdk-angular --save-dev
$ npm install grunt-docular --save-dev
```

How to use the plugin

The Grunt plugin provides a single task, `loopback_sdk_angular`. To use this task in your project's Gruntfile, add a section named `loopback_sdk_angular` to the data object passed into `grunt.initConfig()`. For example:

```
grunt.initConfig({
  loopback_sdk_angular: {
    options: {
      input: '../server/server.js',
      output: 'js/lb-services.js'        // Other task-specific options go here.
    },
    staging: {
        options: {
          apiUrl: '<%= buildProperties.site.baseUrl %>' + '<%=
buildProperties.restApiRoot %>'
        }
    },
    production: {
      options: {
        apiUrl: '<%= buildProperties.site.baseUrl %>' + '<%=
buildProperties.restApiRoot %>'
      }
    }
  }
});
```

Options

Name	Type	Default	Description
`input`	`String`		Path to the main file of your LoopBack server (usually `server.js`).
`output`	`String`		Path to the services file you want to generate, e.g. `js/lb-service.js`.
`ngModuleName`	`String`	`lbServices`	Name for the generated AngularJS module.
`apiUrl`	`String`	The value configured in the LoopBack application via `app.set('restApiRoot')` or `/api`.	URL of the REST API endpoint. Use a relative path if your AngularJS front-end is running on the same host as the server. Use a full URL including hostname when your AngularJS application is served from a different address, e.g. when bundled as a Cordova application.

Example

For example, extend your existing Gruntfile as follows:

1. Add plugin configuration to Gruntfile:

```
grunt.loadNpmTasks('grunt-loopback-sdk-angular');
grunt.loadNpmTasks('grunt-docular');

grunt.initConfig({
  loopback_sdk_angular: {
    services: {
      options: {
        input: '../server/server.js',
        output: 'js/lb-services.js'
      }
    }
  },
  docular: {
    groups: [
      {
        groupTitle: 'LoopBack',
        groupId: 'loopback',
        sections: [
          {
            id: 'lbServices',
            title: 'LoopBack Services',
            scripts: [ 'js/lb-services.js' ]
          }
        ]
      }
    ]
  },
  // config of other tasks
});
```

   For more information about configuring Grunt tasks, see Grunt documentation - Configuring tasks For more about Docular configuration, see grunt-docular.com.
2. Register sub-tasks:

```
grunt.registerTask('default', [
  'jshint',
  'loopback_sdk_angular', 'docular', // newly added
  'qunit', 'concat', 'uglify']);
```

3. Run Grunt to (re)generate files:

```
$ grunt
```

4. Include the generated file in your web application.

```
<script src="js/lb-services.js"></script>
```

5. Start the docular server to view the documentation:

```
$ grunt docular-server
```

Generating Angular API docs

This is currently hidden because of copyright issues with some code in lb-ng-doc. That tool was removed from the SDK.

To generate API documentation for the Angular services, enter this command:

```
$ lb-ng-doc js/lb-services.js
```

The auto-generated source includes ngdoc directives describing all API methods. The comments include as much information from the model definition as possible; for example, the `description` property is copied to the comment, the `accepts` property is converted to a list of input parameters.

Use your favourite `ngdoc` tool to view this documentation; for example, docular.

Example

See the loopback-example-angular for a complete application with a LoopBack server application and an AngularJS client app.

Add the following Gruntfile in the application root directory:

```
module.exports = function(grunt) {
  grunt.initConfig({
    loopback_sdk_angular: {
      services: {
        options: {
          input: 'server/server.js',
          output: 'client/js/lb-services.js'
        }
      }
    },
    docular: {
      groups: [
        {
          groupTitle: 'LoopBack',
          groupId: 'loopback',
          sections: [
            {
              id: 'lbServices',
              title: 'LoopBack Services',
              scripts: [ 'client/js/lb-services.js' ]
            }
          ]
        }
      ]
    }
  });

  // Load the plugin that provides the "loopback-sdk-angular" and "grunt-docular"
tasks.
  grunt.loadNpmTasks('grunt-loopback-sdk-angular');
  grunt.loadNpmTasks('grunt-docular');
  // Default task(s).
  grunt.registerTask('default', ['loopback_sdk_angular', 'docular']);
};
```

Run the gruntfile as follows:

```
$ cd <app-dir>
$ grunt
Running "loopback_sdk_angular:services" (loopback_sdk_angular) task
...
```

This generates the file `client/js/lb-services.js`. Now, run the application:

```
$ node .
```

Xamarin SDK

This project provides early access to advanced or experimental functionality. It may lack usability, completeness, documentation, and robustness, and may be outdated.

However, StrongLoop supports this project: Paying customers can open issues using the StrongLoop

customer support system (Zendesk). Community users, please report bugs on GitHub.

For more information, see StrongLoop Labs.

Introduction

Xamarin enables you to create iOS, Android, and Windows mobile client apps from a single C# codebase.

The LoopBack Xamarin SDK includes a tool that creates a Xamarin C# client API based on a LoopBack server backend API. The client API has a standard set of CRUD methods. In addition, relation methods and custom remote methods in your LoopBack API are generated by the tool as a corresponding C# API. These are documented in Xamarin client API.

There are currently some restrictions on support for custom remote methods. For example, methods with with several return arguments or with multiple body-type parameters are not currently supported. Also, the SDK doesn't currently support custom remote methods that return a custom type not exposed in the API.

Although called Xamarin SDK, this SDK is compatible with any C# application.

Download

Get the SDK from GitHub: https://github.com/strongloop/loopback-sdk-xamarin.

```
$ git clone https://github.com/strongloop/loopback-sdk-xamarin.git
```

A sample application is at https://github.com/strongloop/loopback-example-xamarin.

Folders Overview

The SDK has the following folders:

- `C#` - Source code for the C# part of the generator and testers for the entire generator
  - The open-source of the C# part of the generator, accessed by LBXamarinSDK.sln. This project's end result is `LBXamarinSDKGenerator.dll` that is placed automatically as a post-build event inside the `bin` directory.
- `bin` - contains the `lb-xm` CLI tool.
- `examples` - contains a LoopBack server and a Xamarin solution of an Android app using the SDK, and a second LoopBack server and a Visual Studio solution of a console application using the SDK.
- `lib`
- `test-server` - testers for the lb-xm end result that the Testers act upon.

If you need only a functional SDK generator without source code or examples, you need only the folders lib, bin and the files in the root directory.

Using the SDK generator

Set up

On Windows or MacOS:

```
$ cd loopback-sdk-xamarin
$ npm install
```

By default the SDK generator command will not be installed on your path, so you must either run it from the `lbxamarinsdk-generator/bin` directory or provide the full path to it on the command line.

Generating a Xamarin API

To create a Xamarin client based on a LoopBack API:

1. Create C# code for the SDK; for example:

   ```
   $ cd bin
   $ node lb-xm d:/your-server-app/server/server.js
   ```

   Where `d:/your-server-app/server/server.js` is the full path to your LoopBack application main server script.

   The above command shows an example of running the command directly from the `lbxamarinsdk-generator/bin` directory. If you were running the command from another directory, you would instead use (for example):

   ```
   $ node d:/loopback-sdk-xamarin/bin/lb-xm d:/your-server-app/server/server.js
   ```

   You can also run the `lb-xm` command with options, to (for example) compile a DLL, add compatibility with Xamarin-Forms, and so on. See the command reference below for more information.

2. The generator will save its output in a directory it creates called `output`, in the directory where you run the `lb-xm` command.

lb-xm command

Use the `lb-xm` command to generate a Loopback client in C# based on a LoopBack server API.

The general syntax (when running it in the `lbxamarinsdk-generator/bin` directory) is:

node lb-xm *path-to-server-script* [*options*]

By default the SDK generator command will not be installed on your path, so you must either run it from the `lbxamarinsdk-generator/bin` directory or provide the full path to it on the command line.

ARGUMENTS

path-to-server-script
Full absolute path to the LoopBack application main script. In the standard project layout, `<app-dir>/server/server.js`. Required.

OPTIONS

You can provide one or more of the following options:

`dll`
Compile a dll containing the SDK

`forms`
Ensure compatibility with Xamarin-Forms

`force`
Remove unsupported functions

`check`

Check if the SDK compiles successfully as C# code

For example, the following command generates a C# (`.cs`) file:

```
$ node lb-xm c:/testServer/server/server.js
```

The following command generates a compiled DLL:

```
$ node lb-xm c:/testServer/server/server.js dll
```

The command saves the result in the `bin/output` folder.

Xamarin client API

Setup

You must have a C# development environment, such as Visual Studio or Xamarin Studio. Add to your project the LBXamarinSDK.cs file generated by the lb-xm command.

You must add the following NuGet packages to your project:

- RestSharp Portable
- Newtonsoft JSON

You now have the Xamarin SDK customized for your LoopBack server available in your project.

Overview

The SDK has three entities:

- **Models** - Local instances of the LoopBack models with the same name as the Loopback Models; for example, "Car." They are in the `LBXamarinSDK` namespace.
- **Repositories** - Classes unique to each type of model. They contain the methods that exist on the server: CRUD methods and the custom methods that were added to the Loopback models. Client code uses Repositories to perform server calls. They have the plural name of the Loopback model; for example, "Cars." They are in the `LBXamarinSDK.LBRepo` namespace.
- **Gateway** - A class containing the code that enables the repositories to talk to the server. It is in the `LBXamarinSDK` namespace.

Models

Models in C# have the same name as the Loopback models.

Creating a local instance

The model is a C# class. Use it and create it as you would any other class.

```
Car myCar = new Car()
{
    wheels = 5,
    drivers = 3,
    name = "blarg"
}
```

⚠ Again, this creation is only local. To perform server calls use the repositories.

Repositories

Repositories have the plural name of the Loopback models. They contain the methods that exist on the Loopback server.

For example, if **Car** is the name of the model, then **Cars** is the name of the repository:

```
// Creation of a local instance
Car myCar = new Car();
myCar.accidents = 0;

// Repository method. Persisting local instance to Loopback server
Cars.Create(myCar);
```

The following sections introduce the methods in the repositories.

CRUD methods

The repositories have the same basic model create, read, update, and delete (CRUD) methods for all models, as is in the Loopback server REST API.

⚠ The method createChangeStream() is still unsupported.

Count

```
Task<int> Count(string whereFilter = "")
```

Counts instances of the model that match the specified where filter.

Parameter:

- Optional filter, counting all instances that this filter passes.

Returns the number of matching instances.

For example:

```
int numOfCustomers = await Customers.Count();
int numOfSmartCustomers = await Customers.Count("{\"where\":{\"and\":
[{\"pizzas\":\"500\"}, {\"cats\":\"500\"}]}}");
```

Create

```
Task<T> Create(T theModel)
```

Creates a new instance of the model and persist it into the data source. T is a generic template (the same for all loopback models in the SDK).

Parameter:

- *`theModel`*: The model to create.

Returns the created model.

For example:

```
c = await Customers.Create(c);
```

DeleteById

```
Task DeleteById(String Id)
```

Deletes a model instance by ID from the data source.

Parameter:

- *`Id`* - String ID of the model to delete.

For example:

```
await Customers.DeleteById("1255");
```

Exists

```
Task<bool> Exists(string Id)
```

Checks whether a model instance exists in the data source.

Parameter:

- *`Id`* - the String ID of the model of interest.

Returns a Boolean value.

For example:

```
if(await Customers.Exists("441")) { ... }
```

Find

```
Task<IList<T>> Find(string filter = "")
```

Finds all instances of the model matched by filter from the data source. A filter is applicable here.

Parameter:

- an optional filter to apply on the search.

Returns a list of all found instances.

For example:

```
IList<Customer> l1 = await Customers.Find();
IList<Customer> l2 = await Customers.Find("{\"where\":{\"and\": [{\"pizzas\":\"500\"},
{\"cats\":\"500\"}]}}");
```

FindById

```
Task<T> FindById(String Id)
```

Finds a model instance by ID from the data source.

Parameter:

- *`Id`* - String ID of the model to find.

Returns the found model.

For example:

```
Customer c = await Customers.FindById("19");
```

FindOne

```
Task<T> FindOne(string filter = "")
```

Finds first instance of the model matched by filter from the data source.

Parameter:

- Optional filter to apply on the search.

Returns the found model

For example:

```
Customer c = await Customers.FindOne();
Customer c = await Customers.FindOne("{\"where\":{\"and\": [{\"pizzas\":\"500\"},
{\"cats\":\"500\"}]}}");
```

UpdateAll

```
Task UpdateAll(T updateModel, string whereFilter)
```

Updates instances of the model matched by where from the data source.

Parameters:

- *`updateModel`* - Update pattern to apply to all models that pass the where filter
- *`whereFilter`* - Where filter.

For example:

```
Customer updatePattern = new Customer() {cars = 99};
await UpdateAll(updatePattern, "{\"where\":{\"and\": [{\"pizzas\":\"500\"},
{\"cats\":\"500\"}]}}");
```

this updates all customers who have 500 cats and 500 pizzas to have 99 cars.

UpdateById

Task<T> UpdateById(String *Id*, T *update*)

Updates attributes of a model instance and persists it to the data source.

Parameters:

- *`Id`* - String ID of the model to update
- *`update`* - the update to apply.

Returns the updated model.

For example:

```
Customer c = new Customer() { hp = 100 };
h = await UpdateById("11", c);
```

There are more methods created automatically if relations between models are added. Their definition and description are in the generated code.

Upsert

Task<T> Upsert(T *theModel*)

Updates an existing model instance or insert a new one into the data source.

Parameter:

- *`theModel`*, the model to update/insert.

Returns the updated/inserted model.

For example:

```
c = await Customers.Upsert(c);
```

Relation Methods

The SDK will create C# code for methods that are created for models connected by relations, just like in the REST API.

Example

Suppose we have two Loopback models: Goal and TodoTask, and the relation between them is that TodoTask belongs to Goal.

Then the SDK will create, among others, the method to fetch the goal of a given todo task. You can see the generated code in the TodoTasks repository:

```
/*
 * Fetches belongsTo relation goal
 */
public static async Task<Goal> getGoal(string id, [optional parameters])
{
...
}
```

Then a usage example is:

```
var TodoTaskID = "10034";
Goal relatedGoal = await TodoTasks.getGoal(TodoTaskID);
```

Custom remote methods

The LoopBack Xamarin SDK supports custom remote methods and will create C# code for them. These methods exist in the repository classes.

Example

Suppose there is a LoopBack model called Logic. Then, if there is a custom method in the LoopBack application in **`Logic.js`** as follows:

```
module.exports = function(Logic) {
    Logic.determineMeaning = function(str, cb) {
        console.log(str);
        cb(null, 42);
    };

    Logic.remoteMethod(
        'determineMeaning',
        {
            accepts:  {arg: 'str', type: 'string', http: { source: 'body' }, required:
true},
            returns: {arg: 'res', type: 'number', root: true},
            http: {path: '/determineMeaning', verb: 'post'},
            description: 'This is the description of the method'
        }
    );
};
```

Then once we compile an SDK using lb-xm, a method will be created in the C# code of **LBXamarinSDK.cs**, with the signature:

```
Task<double> determineMeaning(string str)
```

If you go into the *Logic*s repository code you can see the generated code for the method:

```
/*
 * This is the description of the method
 */
public static async Task<double> determineMeaning(string str)
{
...
}
```

Then an example of using it:

```
double something = await Logics.determineMeaning("blarg");
```

Gateway

Gateway has several important methods.

isConnected

Task<bool> isConnected(int *timeoutMilliseconds* = 6000)

Checks if the server is connected.

Parameter:

- *timeoutMilliseconds* - timeout for the check, in ms.

Returns a Boolean value.

```
connectedStatus = await Gateway.isConnected();
```

There are additional Gateway methods with a descriptions in the generated code.

SetAccessToken

void SetAccessToken(AccessToken *accessToken*)

All server calls will be done with the authorization given by the supplied access token.

```
var list1 = await Users.Find(); // error 401, not authorized
AccessToken token = await Users.login(auth); // Server call: Performing login
Gateway.SetAccessToken(token);
var list2 = await Users.Find(); // ok, list is returned
```

SetDebugMode

This method sends output when important SDK functionality is executed (output from `System.Diagnostics.Debug`) to the console.

void SetDebugMode(bool *isDebugMode*)

Once debug mode is set to true, important information will be sent to the debug console.

For example, if we make a server call:

```
Customer myCustomer = new Customer()
{
   name = "joe",
   geoLocation = new GeoPoint()
   {
      Longitude = 0,
      Latitude = 11
   }
};
myCustomer = await Customers.Create(myCustomer);
```

Then when the debug mode is on, calling the above code will display the following output to the console:

```
-------- >> DEBUG: Performing POST request at URL:
'http://10.0.0.1:3000/api/Customers', Json:
{"geoLocation":{"lat":11.0,"lng":0.0},"name":"joe"}
```

SetServerBaseURL

void SetServerBaseURL(Uri *baseUrl*)

Parameter:

- baseUrl - URL of the server API.

Sets the API URL of the loopback server. For example:

```
SetServerBaseURL(new Uri("http://10.0.0.1:3000/api/"));
```

SetTimeout

void SetTimeout(int *timeoutMilliseconds* = 6000)

Sets the timeout of server calls. if no response is received from server after this period of time, the call will die.

```
Gateway.SetTimeout(2000);
```

Handling exceptions

Perform all access to a repository (calls to methods that communicate with the server) inside a `try / catch` clause, including a clause to catch a `RestException`.

`RestException` has a StatusCode field that specifies the return status of the error.

For example:

```
User myUser = new User()
{
  email = "j@g.com",
  password = "123"
};
try
{
 myUser = await Users.Create(myUser); // user created on the server. success.
}
catch(RestException e)
{
 if(e.StatusCode == 401)
 {
    // not authorised to create the user
 }
 else if(e.StatusCode == 422)
 {
    // unprocessable entity. Perhaps the email already exists, perhaps no password
    // is specified inside the model myUser, etc
 }
}
```

Xamarin example app

This article is reproduced from loopback-example-xamarin

LoopBack Xamarin SDK example application

This repository contains a mobile app that demonstrates the Loopback Xamarin SDK. It contains:

- `Server` directory: Loopback server application.
- `Client` directory: The ToDo client app, created in Xamarin Studio with Xamarin Forms for iOS.

Prerequisites

- Install Xamarin Studio. **NOTE**: You must get the full Studio (trial) license, not just the “starter” license.
- Install Xamarin Forms
- Install LoopBack

Download

```
$ git clone https://github.com/strongloop/loopback-example-xamarin.git
```

Run the server app

You can either run the LoopBack server app yourself, or connect to the demo app running at http://xamarindemo.strongloop.com.

Run your own server app

1. Go to server folder:
   `$ cd server`
2. Install dependencies:

```
$ npm install
```

3. Run the application:
```
$ node .
```

Use StrongLoop's server app

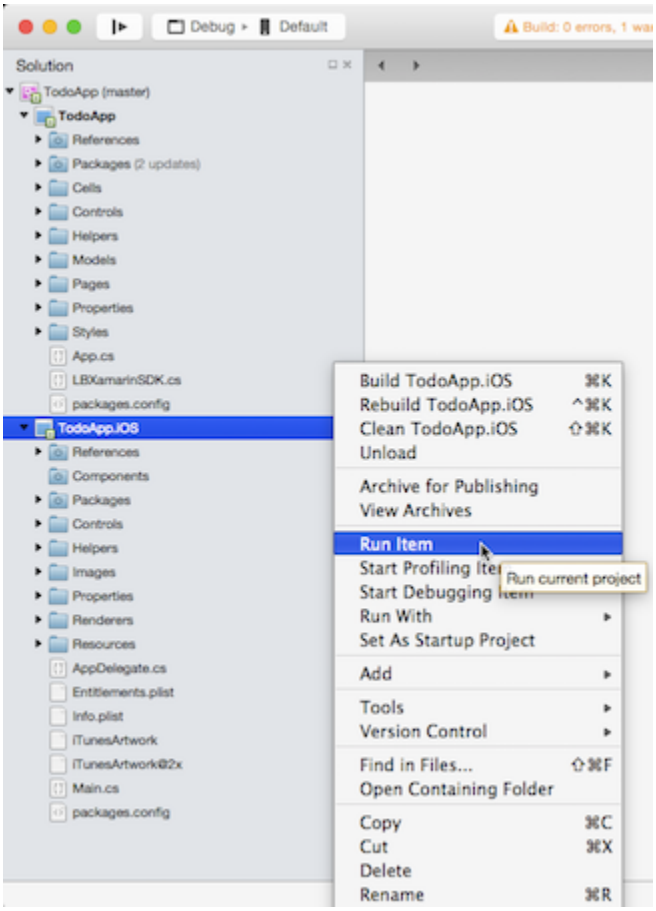
Alternatively, you can run the Xamarin client app against http://xamarindemo.strongloop.com.

Edit LBXamarinSDK.cs and change BASE_URL to `http://xamarindemo.strongloop.com/api`.

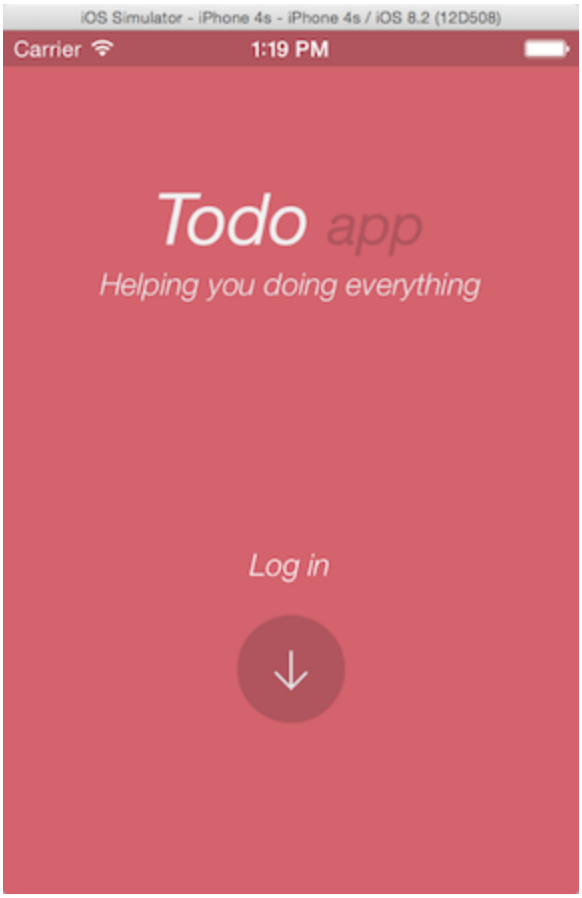
Run the client app

Open the client app solution with Xamarin Studio: `loopback-example-xamarin/Client/Todo App/TodoApp.sln`.

In Xamarin Studio, build and run the app:



The iOS Simulator will appear running the client app. It may take a moment or two.

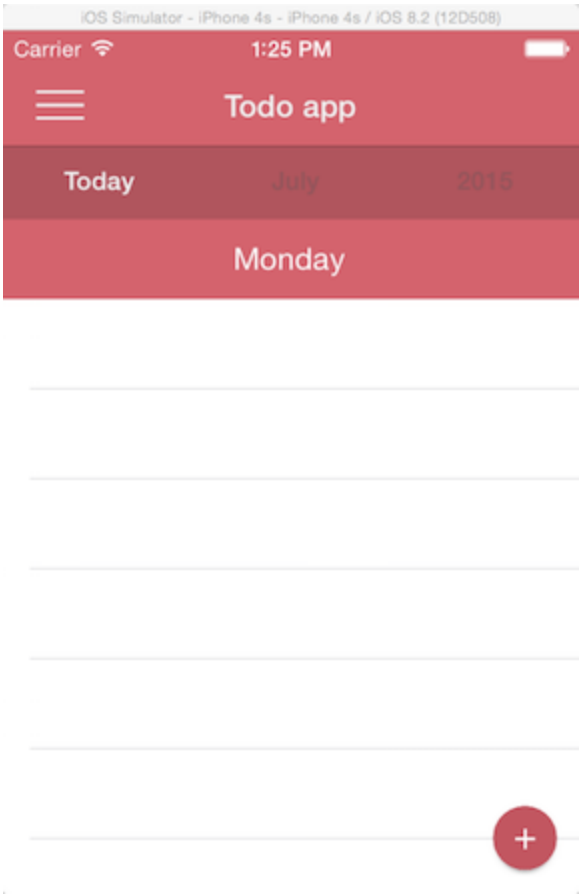


In the iOS Simulator:

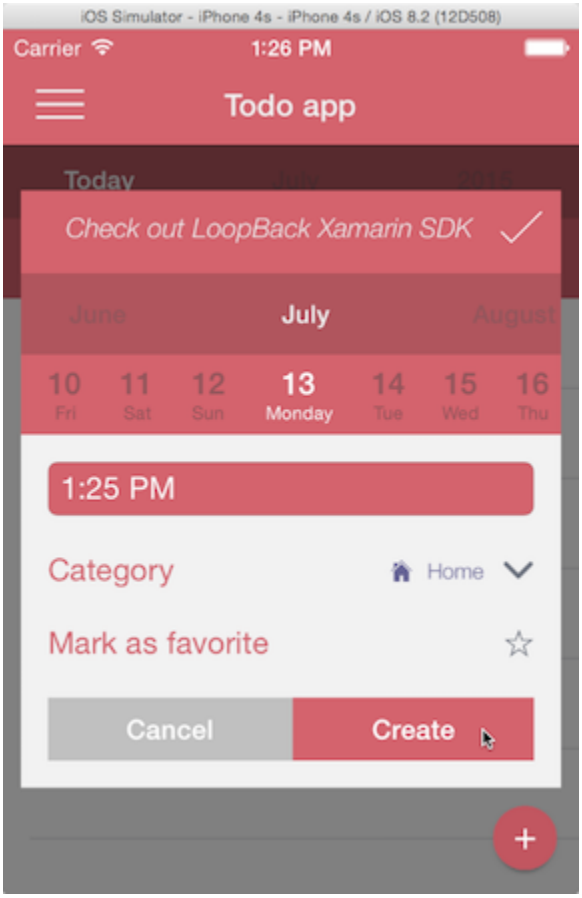
1. The first time you run the app, click **Sign Up**:

- Enter an email address and password (they are not validated).
- Click **Sign Up**
- If you already signed up in this session, click **Login** with the credentials you entered previously.

1. You'll see the app home page:

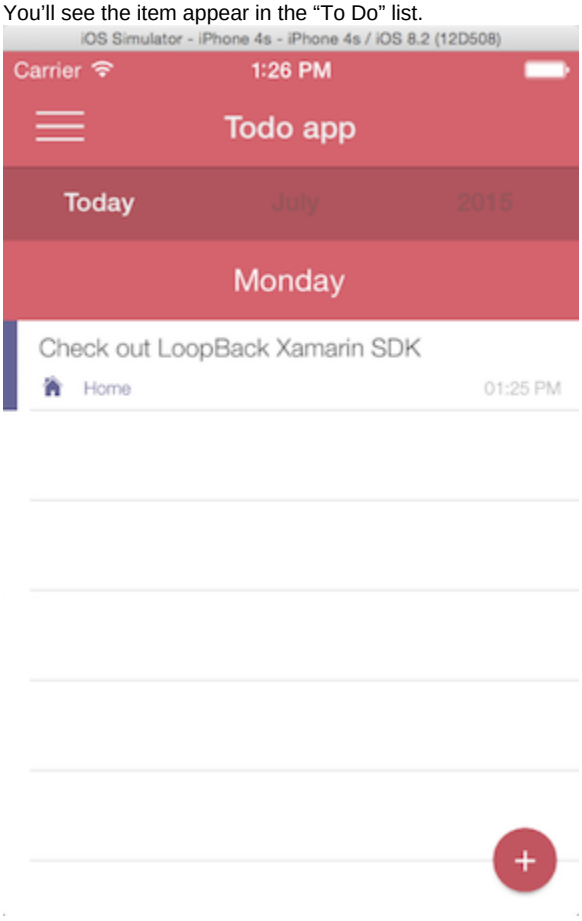


2. Enter a “To Do” item:



- Click **+**
- Click **Add a task + **
- Enter a description of the task in the top (red) text field
- Change the date, time, category and mark as a favorite if you wish.
- Click **Create** to add the item.

You'll see the item appear in the "To Do" list.



You can confirm that the data is also added to the LoopBack model using LoopBack API Explorer.

Links

- LoopBack
- Perfected Tech
- Xamarin Studio

LoopBack in the client

StrongLoop Labs

This project provides early access to advanced or experimental functionality. It may lack usability, completeness, documentation, and robustness, and may be outdated.

However, StrongLoop supports this project: Paying customers can open issues using the StrongLoop customer support system (Zendesk). Community users, please report bugs on GitHub.

For more information, see StrongLoop Labs.

LoopBack in the client is sometimes referred to as *isomorphic LoopBack*, because it provides the exact same API as the LoopBack server framework.

To run LoopBack in the browser, you must use Browserify. You can create models and use the memory adapter and have your LoopBack app fully running in the browser. To further extend this, you can seamlessly connect models using the Remote Connector to connect LoopBack to LoopBack. For example, browser to server or server to server.

This capability is implemented in strong-remoting which can accept and return ModelTypes in addition to the JSON and JSON primitives. Because models don't need to know where they are being run other than in LoopBack, you can share the same API regardless of where your code is running, thus making the API isomorphic.

Models are referenced through one way, "local" versus "remote" as seen in loopback-example-offline-sync. This is the foundation of the replication API for data synchronization between browser and server.

Running LoopBack in the browser

In the browser, the main application file must call the function exported by the `loopback-boot` module to setup the LoopBack application by executing the instructions contained in the browser bundle:

browser-app.js

```
var loopback = require('loopback');
var boot = require('loopback-boot');

var app = module.exports = loopback();
boot(app);
```

The app object created above can be accessed via `require('loopback-app')`, where `loopback-app` is the identifier used for the main app file in the Browserify build shown above.

Here is a simple example demonstrating the concept:

index.html

```
<script src="app.bundle.js"> </script>
<script>
  var app = require('loopback-app');
  var User = app.models.User;
  User.login({
    email: 'test@example.com',
    password: '12345'
  }, function(err, res) {
    if (err) {
      console.error('Login failed: ', err);
    } else {
      console.log('Logged in.');
    }
  });
</script>
```

Using Browserify

Use Browserify to create a LoopBack API in the client.

The build step loads all configuration files, merges values from additional config files like `app.local.js` and produces a set of instructions that can be used to boot the application.

These instructions must be included in the browser bundle together with all configuration scripts from `models/` and `boot/`.

Don't worry, you don't have to understand these details. Just call `boot.compileToBrowserify()`, and it will take care of everything for you.

Build file (Gruntfile.js, gulpfile.js)

```
var browserify = require('browserify');
var boot = require('loopback-boot');

var b = browserify({
  basedir: appDir,
});

// add the main application file
b.require('./browser-app.js', { expose: 'loopback-app' });

// add boot instructions
boot.compileToBrowserify(appDir, b);

// create the bundle
var out = fs.createWriteStream('browser-bundle.js');
b.bundle().pipe(out);
// handle out.on('error') and out.on('close')
```