German University in Cairo
Faculty of Media Engineering and Technology
Prof. Slim Abdennadher & Dr. Hisham Othman,
Eng. Ramy Aboul Naga & Eng. Youmna Bassiouny.

CSEN 704/DMET 706, Advanced Computer/Media Lab
Winter 2014

Integration; eShop

Submission Week: **29/11/2014**

In this exercise, we will walk through building an connecting an Android application to a Ruby on Rails backend.

Starting the web application

1. Instead of having to setup the Ruby on Rails environment, we're going to use a Web IDE to complement an already existing Rails application with API-handling logic, to make it mobile-accessible. The Web IDE we're going to use is Cloud9's Web IDE.
   a. Go to http://c9.io/ to sign up, then go to your dashboard.
   b. Create a new Workspace, and call it "eShop_XX_XXXXX", replacing XX_XXX with your GUC ID.
      i. Workspace Privacy: Open and Discoverable
      ii. Hosting: Hosted
      iii. Technology: Custom
   c. Wait for the workspace to finish processing.
   d. When it's processed, reload the page then choose click on its name in the left sidebar, under "My Projects".
   e. Click on "Start Editing" to launch the Web IDE. You'll be presented with a page divided into 3 panes: Workspace Explorer (on the left), File Editor (in the middle), and Terminal (at the bottom).
2. Download "[Integration] eShop.zip" from the course's page on the MET website to your Desktop, and extract it.
   a. Open up "Eclipse ADT", and import the Android project within the "mobile" folder in the archive you just extracted.
   b. In the Web IDE, Click on File → Upload Local Files…, then drag the archive "mobile.tgz" from the extracted archive on your Desktop to where it says "DRAG & DROP" in the IDE.
   c. After the file is uploaded, click on "Close".
3. In the Terminal page, run `tar -zxvf web.tgz && mv web/* . && rm -r web.tgz web README.md` to extract the archive you just uploaded.
4. Run `bundle install` to install the application's dependencies.
5. Run `sudo service mysql start` to start the MySQL server, then run `rake db:create db:migrate` to create and migrate the database.
6. To prefill the database with some sample data, run `mysql -u root -D eShop <data/products.sql`.
7. Now all is set, click on Run → Run With → Ruby on Rails (Rails 4.0 + Ruby 2.1) and wait for the server to boot up, then click on Preview → Preview with Web Server to open the application's URL in a new tab.

Building the API

1. Open up config/routes.rb, and before the call to `root`, add the following lines. The `defaults: { format: :json }` option tells Rails that there's no need to append .json to the routes within the

`:api` namespace for the routes to work, since it will be set by default.

```
# API
namespace :api, defaults: { format: :json } do
  resources :sessions, only: :create

  resources :products, only: :index do
    member do
      patch :buy
    end
  end
end
```

2. In app/controllers/api, add products_controller.rb with the following content. The `Api::` before the controller's name says that it belongs to the `:api` namespace. The `Api::ProductsController` inherits from a `Api::BaseController`, which contains logic common to all of the API controllers.

```
class Api::ProductsController < Api::BaseController
  before_action :authenticate_user!, only: :buy

  def index
    respond_with @products = Product.all
  end

  def buy
    @product = Product.find(params[:id])
    @product.buy_by(current_user)

    respond_with @product
  end
end
```

3. The logic for handling authentication in the API differs a bit from that used in the frontend controllers. We need to override some of the methods for it to work properly. In the `Api::BaseController`, add the following lines.

```
class Api::BaseController < ApplicationController
  skip_before_action :verify_authenticity_token

  respond_to :json

protected

  def current_user
    @current_user ||= User.find_by(token: request.headers[:authorization])
  end

  def authenticate_user!
    unless current_user?
      render status: :unauthorized
    end
  end
end
```

   a. `respond_to :json` tells the controller to only respond to JSON formats, which we set as the default in the routes.
   b. `User.find_by(token: request.headers[:authorization])` tries to get the user using the token set in the `HTTP_AUTHORIZATION` header.

c. `render status: :unauthorized` responds with a status code instead of redirecting the user to the `root_path`.

4. Going to /api/products now should display a list of the products in JSON. This is the list that the mobile application will use as its source of data for the products.
5. Create a sessions_controller.rb in app/controllers/api for validating the user's credentials. Add the following to it.

```
class Api::SessionsController < Api::BaseController
  def create
    respond_with @user = User.authenticate(*session_params.values_at(:email,
:password))
  end

protected

  def session_params
    params.require(:session).permit(:email, :password)
  end
end
```

6. Authentication differs in the API in that a uniquely generated token is assigned to each user, with which private requests are made. This token is used to identify the user, since the requests are stateless, and APIs don't work with sessions. Run `rails g migration add_token_to_users token` to add a `token` column to the `users` table, then run `rake db:migrate` to migrate the database.
7. The token should be generated whenever a user registers. The following will assign a random hexadecimal string as the user's token before creating his/her record in the users table. Make sure to add it to the `User` model.

```
# Callbacks
before_create -> { self.token = SecureRandom.hex }, unless: :token?
```

8. Now go to /users/new in the browser, and register a new user.
9. After registering, open a new Terminal tab and run `rails c`. When it loads, type `User.last.token` to check the token that was generated for you.
10. To customize the returned JSON after logging in through the API, create app/views/api /sessions/create.json.rabl and add the following to it.

```
object @user => nil
attributes :id, :name, :email, :token
```

11. For this to work, add `gem 'rabl'` to the `Gemfile` and run `bundle`. Remember to restart the server for the changes to be picked up.

Android, meet the API

1. The Android project you downloaded includes many libraries that facilitate the task of connecting to APIs, such as Retrofit, Gson, and Picasso. Read more about each of them to know how to utilize their strength to your will.
2. Make sure the project builds successfully on your device before proceeding. When launched, a login form should be displayed on the emulator/device.
3. Make the application aware of the available API routes by defining them in one of two places, Public or Private, based on the access level they need.
   a. Grant the application access to the Internet by adding the following to AndroidManifest.xml after the `<users-sdk />` tag.

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

   b. Configure the base URL of the API in util/ApiRouter.java.

c. Public Access, in util/PublicApiRoutes.java.

```
public interface PublicApiRoutes {
	@POST("/sessions")
	@FormUrlEncoded
	void login(@Field("session[email]") String email,
@Field("session[password]") String password,
			Callback<User> callback);

	@GET("/products")
	void getProducts(Callback<List<Product>> callback);
}
```

d. Private Access, in util/PrivateApiRoutes.java, which requires a token to be passed in the `HTTP_AUTHORIZATION` header.

```
public interface PrivateApiRoutes {
	@PATCH("/products/{product_id}/buy")
	void patchProductBuy(@Path("product_id") long productId,
Callback<Response> callback);
}
```

e. This behavior is defined in util/ApiRouter.java and util/PrivateApiInterceptor.java. The Retrofit library is what makes this all possible.

4. Similar to the Rails applications, a good design is to create models in your Android application that have the same properties of your Rails models. Create the following files with the corresponding content.

a. model/Product.java.

```
public class Product {
	private long id;
	private String name;
	private double price;
	private String imageUrl;
	private int stock;
}
```

b. model/User.java.

```
public class User {
	private long id;
	private String name;
	private String email;
	private String token;
}
```

c. Use Eclipse's Source Generator to generate setters and getters for the private fields.

d. Make sure to uncomment the lines in activity/base/BaseActivity.java and activity/ProductsActivity.java that use the User and Product models' setter and getters.

5. In activity/ProductsActivity.java, replace the `refreshViews()` method with the following code to invoke the API, retrieve a list of products, and display them on the screen.

```
protected void refreshViews() {
	super.refreshViews();

	adpProducts.clear();

	startProgress();

	ApiRouter.withoutToken().getProducts(new Callback<List<Product>>() {
		@Override
		public void success(List<Product> products, Response response) {
			adpProducts.addAll(products);
			stopProgress();
		}

		@Override
		public void failure(RetrofitError e) {
			displayError(e);
		}
	});
}
```

6. In activity/LoginActivity.java, add the following code directly after the call to `Log.d(…)` to handle validating the user's credentials. Replace the `TODO` with `referrer`'s `if` condition, that launches the activity that requested authentication, after the user's credentials are validated successfully. In our case, this is the ProductsActivity.

```
ApiRouter.withoutToken().login(email, password, new Callback<User>() {
	@Override
	public void success(User user, Response response) {
		setCurrentUser(user);

		stopProgress();

		// TODO: Start the referrer activity
	}

	@Override
	public void failure(RetrofitError e) {
		displayError(e);

		btnLogin.setEnabled(true);
	}
});
```

7. The current progress should allow you to test the application.

8. To allow users to purchase products, add the following to the `onClick()` method of the `btnBuy`'s `onClickListener`.

```
@Override
public void onClick(View v) {
	startProgress();

	ApiRouter.withToken(getCurrentUser().getToken()).patchProductBuy(product.getI
d(),
			new Callback<Response>() {
				@Override
				public void success(Response response, Response
rawResponse) {
					Toast.makeText(ProductsActivity.this, "Bought: " +
product.getName(),
							Toast.LENGTH_LONG).show();

					stopProgress();

					product.setStock(product.getStock() - 1);
					adpProducts.notifyDataSetChanged();
				}

				@Override
				public void failure(RetrofitError e) {
					displayError(e);
				}
			});
}
```

And that's it; the Android application is now fully connected to the Rails application. Make sure to read the rest of the application's code to learn more about the integration process.