Давайте создадим наше последнее в этом курсе и самое грандиозное приложение — будем отображать погоду. Это же приложение мы выгрузим в Google Play в конце курса.

В первую очередь нам нужно какое-то бесплатное API для получения прогноза. Лучший на мой взгляд вариант — [OpenWeatherMap](https://openweathermap.org/). [Зарегистрируйтесь](https://home.openweathermap.org/users/sign_up) на этом сервисе и получите [ключ API](https://home.openweathermap.org/api_keys).

**OpenWeatherMap** даёт 60 бесплатных запросов к API в минуту. Этого вполне достаточно для начала — если вы превысите количество запросов, то, я думаю, вы вполне сможете позволить себе платный тариф — он банально окупится рекламой.

Retrofit

Самый простой и распространённый, но при этом невероятно гибкий и мощный способ работы с API в Android — это библиотека [Retrofit](https://square.github.io/retrofit/).

Добавим её в build.gradle:

implementation 'com.squareup.retrofit2:retrofit:2.3.0'

Retrofit, помимо прочего, работает не только с JSON, но и со многими другими форматами. Начиная с версии **2.0**парсинг различных форматов вынесен в отдельную сущность — конвертеры и каждый конвертер вынесен в отдельную библиотеку.

По этой причине нам будет нужно также подключить библиотеку для работы с JSON конвертером (в нашем случае — с использованием библиотеки GSON):

implementation 'com.squareup.retrofit2:converter-gson:2.3.0'

Не забудьте добавить разрешение для доступа к интернету в манифест

Модели в Retrofit

В лучших традициях Android, API в Retrofit описывается декларативным образом — мы создаём декларативное описание моделей, которые будем получать из API, используя POJO — простые Java-объекты, и интерфейс, сообщающий, какие запросы мы будем отправлять.

Наверное, сейчас не очень понятно — давайте попробуем на практике.

Нас интересуют два запроса — текущая погода и прогноз на 5 дней. Начнём с первого:

https://api.openweathermap.org/data/2.5/weather?lat=35&lon=139&appid=ВАШ\_КЛЮЧ\_АПИ&units=metric

[](https://skillberg.com/media/uploads/2017/11/17/7fa2ef0c5ea542a2a992e4c682702522.png)

Что нас интересует в данном ответе?

1. Поле main.
2. Поле wind.
3. Поле name.

Пока этого достаточно.

Создадим модели, начём с main:

public class ForecastMain {

private float temp;

private float pressure;

private float humidity;

@SerializedName("temp\_min")

private float minTemp;

@SerializedName("temp\_max")

private float maxTemp;

public float getTemp() {

return temp;

}

public float getPressure() {

return pressure;

}

public float getHumidity() {

return humidity;

}

public float getMinTemp() {

return minTemp;

}

public float getMaxTemp() {

return maxTemp;

}

}

Аннотация SerializedName позволяет указать имя поля в JSON, из которого будет десериализоваться поле в Java-объекте.

Теперь модель для поля wind:

public class ForecastWind {

private float speed;

@SerializedName("deg")

private float degree;

public float getSpeed() {

return speed;

}

public float getDegree() {

return degree;

}

}

Теперь создадим объект, который будет отображать "корневую" структуру JSON-ответа и содержать нужные нам поля:

public class CurrentWeather {

private ForecastMain main;

private ForecastWind wind;

@SerializedName("name")

private String cityName;

public ForecastMain getMain() {

return main;

}

public ForecastWind getWind() {

return wind;

}

public String getCityName() {

return cityName;

}

}

Интерфейс API

Теперь нужно создать интерфейс, который будет описывать API, к которому мы хотим получить доступ.

public interface Api {

@GET("weather")

Call<CurrentWeather> getCurrentWeather(

@Query("lat") double latitude,

@Query("lon") double longitude,

@Query("appid") String apiKey,

@Query("units") String units

);

}

Пока что в нём лишь один метод, получающий текущую погоду. Давайте разберём его.

В первую очередь обратите внимание на аннотацию @GET. Она говорит о том, какой HTTP-метод будет использоваться — в нашем случае это, очевидно, GET. Возможные варианты:

* GET
* POST
* PUT
* PATCH
* HEAD
* DELETE
* OPTIONS

В качестве единственного параметра передаётся путь до метода: мы используем URL https://api.openweathermap.org/data/2.5/weather, в этом случае путь — weather.

Параметры метода будут переданы в HTTP-запросе. В нашем случае это query-параметры, которые будут переданы прямо в URL.

Параметр аннотации @Query — название параметра для запроса.

Метод возвращает Generic-класс Call, который используется для выполнения запроса. Тип дженерика — это класс, объект которого мы хотим получить после выполнения запроса, то есть те десериализованные данные, которые придут нам в ответ на запрос.

Инстанциируем API

Последнее, что осталось сделать — создать объект, с помощью которого мы будем выполнять запросы к API.

Создадим для этого класс со статическим методом:

public class ApiFactory {

public static Api createApi() {

Retrofit retrofit = new Retrofit.Builder()

.baseUrl(Constants.API\_BASE\_URL) // Базовый URL

.addConverterFactory(GsonConverterFactory.create()) // Конвертер JSON

.build();

return retrofit.create(Api.class);

}

}

Константу API\_BASE\_URL я вынес в отдельный класс, туда же я вынес ключ API и дефолтную единицу измерения:

public class Constants {

public static final String API\_BASE\_URL = "https://api.openweathermap.org/data/2.5/";

public static final String API\_KEY = "ac4d0e2053d938b703a7a3c24806fb27"; // Замените ключ на свой

public static final String DEFAULT\_UNITS = "metric";

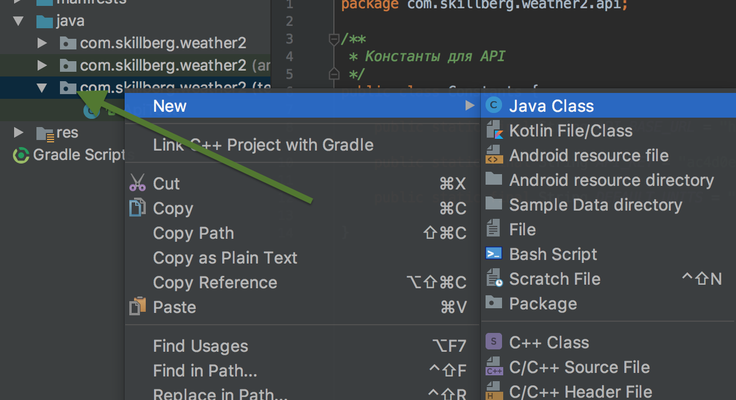
}

Тестируем API

Как бы нам теперь проверить работоспособность того, что мы написали? В этом нам помогут Unit-тесты!

Если вы никогда не использовали тесты — почитайте про них, например, [тут](http://www.javenue.info/post/19).

В проекте уже есть пакет с тестами. Создайте там новый класс с тестами API:

[](https://skillberg.com/media/uploads/2017/11/17/3f556f6e414a42e5b708a01a458abd97.png)Создание нового теста

public class ApiTest {

private final Api api = ApiFactory.createApi();

}

В нём мы пока что только создали объект API.

Теперь создадим метод, который будет тестировать:

@Test

public void testCurrentWeather() throws IOException {

}

В нём создадим запрос — Call:

Call<CurrentWeather> call = api.getCurrentWeather(

35, // широта

139, // долгота

Constants.API\_KEY,

Constants.DEFAULT\_UNITS);

На данный момент мы только создали запрос, но ещё не отправили его. Существует два способа отправки запроса:

* **Синхронный** — будет выполнен в том же потоке, где и вызван
* **Асинхронный** — будет выполнен в фоне, после чего на том потоке, откуда вызван запрос будет вызван коллбэк.

Вообще, в приложениях обычно используется асинхронный способ, но для тестов нам понадобится синхронный.

Запрос выполняется очень просто:

Response<CurrentWeather> response = call.execute();

Если произойдёт ошибка, связанная с сетью (например, сети не будет), то метод execute() бросит IOException.

Если же произойдёт HTTP-ошибка (например, 400 или 500), то ответ будет отмечен как неуспешный. В первую очередь проверим это:

Assert.assertTrue(response.isSuccessful());

Теперь получим тело ответа:

CurrentWeather body = response.body();

Тело ответа может быть null, поэтому неплохо было бы это проверить:

Assert.assertNotNull(body);

Теперь просто получим из тела ответа имя города, и проверим, что оно присутствует:

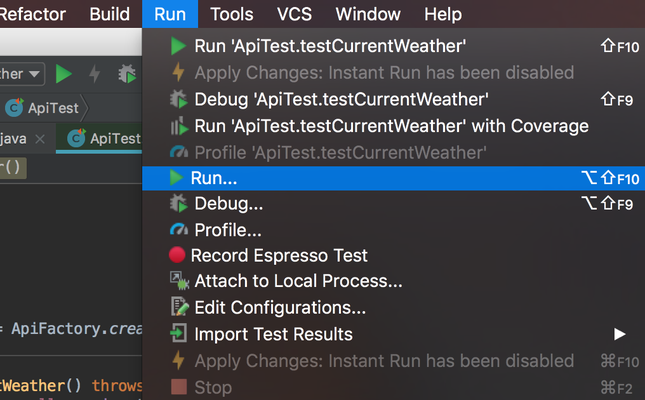
String cityName = body.getCityName();

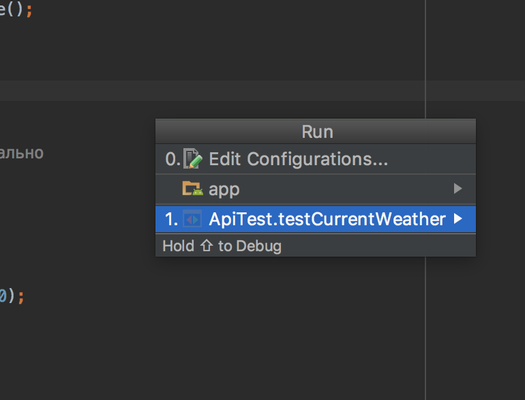
Assert.assertNotNull(cityName);

Assert.assertTrue(cityName.length() > 0);

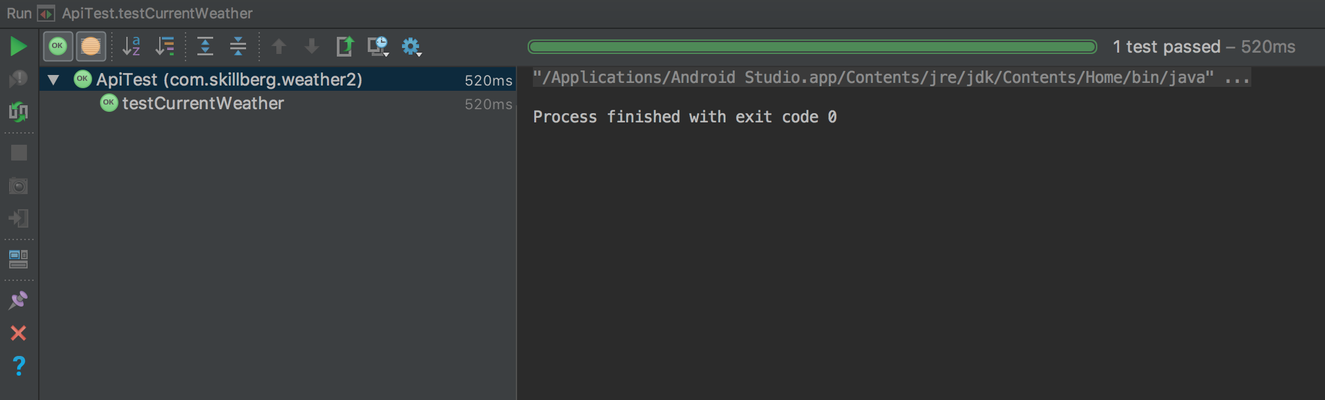
Точно так же мы можем получить и все остальные данные. Попробуйте сделать это сами, а если не получится — можете подглядеть в исходниках проекта :)

Попробуйте запустить тесты, например, вот так:

[](https://skillberg.com/media/uploads/2017/11/17/c39957cca57e4fc2a247405b0dd6bb51.png)Запуск теста

[](https://skillberg.com/media/uploads/2017/11/17/0a5f36b0994e4413b41ea850f17bb02f.png)Запуск теста

После того, как тест выполнится, вы увидите результаты — в нашем случае успешные:

[](https://skillberg.com/media/uploads/2017/11/17/882f7d4f19404866a983231f07bba5f0.png)Результаты выполнения теста

На этом пока всё. А в следующем уроке мы с вами научимся работать с геолокацией!

ПРИМЕРЫ КОДА

[ИСХОДНЫЙ КОД УРОКА](https://github.com/Skillberg/WeatherAndroid2/tree/lesson-32)

ДОМАШНЕЕ ЗАДАНИЕ

Допишите тест для проверки всех остальных данных.