**How to create REST API for Android app using PHP, Slim and MySQL – Day 1/2**

by [Ravi Tamada](http://www.androidhive.info/author/admin/)/ January 19, 2014/ [**92** Comments](http://www.androidhive.info/2014/01/how-to-create-rest-api-for-android-app-using-php-slim-and-mysql-day-12-2/#disqus_thread)

If you are going to build an android application (it can be any other mobile platform or web too) that manages all the user data on a central database, REST API will be good architectural option to do the communication between the app and the server.

If you consider [Evernote](https://play.google.com/store/apps/details?id=com.evernote), [Wunderlist](https://play.google.com/store/apps/details?id=com.wunderkinder.wunderlistandroid) apps, these apps can uninstalled at anytime and once we install them back and login, all our data will be restored. This is because all the data will stored in a cloud database and communication b/w app and database will be done using a REST API.

This tutorial gives enough knowledge about building a REST API for very beginners. As this tutorial seems lengthy, I had divided it into 2 parts. In the 1st part we learn fundamental concepts of REST and do the required setup. In the 2nd part building actual API (writing PHP & MySQL code) is covered.

[DOWNLOAD CODE](http://download.androidhive.info/download?code=Af5zV3Fmd8DKUL4fkN66pbPXsEO5oiaWabC%2BZ0Ekem3cdd48fb1710IE%2B80r2vLVAFkr1HX7preT9Q0W4toErT3nC5MwXOW3vIXD%2BelsMUltp0t9Bm%2ByFvKqA%3D%3DzxEDnj3DwtbbqLmQ484q9suQqfzVv6OacCe)

VIDEO DEMO

**1. Basics of REST API Design**

REST architecture will be useful to build client/server network applications. REST represents Representational State Transfer. Implementing REST is very simple compared to other methods like SOAP, CORBA, WSDL etc., It basically works on HTTP protocol.



Following are the list of things should be considered while building a REST api.

**» HTTP Methods**  
A well-designed RESTful API should support most commonly used HTTP methods (GET, POST, PUT and DELETE). There are other HTTP methods like OPTIONS, HEAD but these are used most often. Each method should be used depending on the type of operation you are performing.

|  |  |
| --- | --- |
| **GET** | To fetch a resource |
| **POST** | To create a new resource |
| **PUT** | To update existing resource |
| **DELETE** | To delete a resource |

**» HTTP Status Code**  
HTTP status codes in the response body tells client application what action should be taken with the response. For an example if the response code 200, it means on the server side the request is processed successfully and you can expect updated data in the response. As well if the status code is 401, the request is not authorized. An example cause for 401 could be api key is invalid.

It is not necessary to support all HTTP status codes, but supporting at least the following codes should be good enough. Check out list of http codes from [restapitutorial.com](http://www.restapitutorial.com/httpstatuscodes.html) and [Wikipedia](http://en.wikipedia.org/wiki/List_of_HTTP_status_codes)

|  |  |
| --- | --- |
| **200** | OK |
| **201** | Created |
| **304** | Not Modified |
| **400** | Bad Request |
| **401** | Unauthorized |
| **403** | Forbidden |
| **404** | Not Found |
| **422** | Unprocessable Entity |
| **500** | Internal Server Error |

**» URL Structure**  
In REST design the URL endpoints should be well formed and should be easily understandable. Every URL for a resource should be uniquely identified. If your API needs an API key to access, the api key should be kept in HTTP headers instead of including it in URL.

For an example:  
GET http://abc.com/v1/tasks/11 – Will give the details of a task whose id is 11

POST http://abc.com/v1/tasks – Will create a new task  
  
**» API Versioning**  
There is a huge [discussion](http://stackoverflow.com/questions/389169/best-practices-for-api-versioning) on API versioning whether to maintain api version in the URL or in the HTTP request headers. Even though it is recommended that version should be included in the request headers, I feel comfortable to maintain it in the URL itself as it is very convenient on the client side to migrate from one version to another.

Example:  
http://abc.com/**v1**/tasks  
http://abc.com/**v2**/tasks  
  
**» Content Type**  
The Content Type in HTTP headers specifies the kind of the data should be transferred between server and client. Depending upon the data your API supporting you need to set the content type.

For an example, JSON Mime type should be **Content-Type: application/json**, for XML **Content-Type: application/xml**. You can find list of supported MIME Types [here](http://en.wikipedia.org/wiki/MIME_type)  
  
**» API Key**  
If you are building a private API where you want to restrict the access or limit to a private access, the best approach is to secure your API using an API key. This article [Designing a Secure REST (Web) API without OAuth](http://www.thebuzzmedia.com/designing-a-secure-rest-api-without-oauth-authentication/) by Riyad Kalla covers the best way to secure you rest api. But as this article aims at very beginners I am not going with any complex model. So for now we can go with generating a random api key for every user. The user is identified by the api key and all the actions can be performed only on the resources belongs to him.

The API key should be kept in request header **Authorization** filed instead of passing via url.

**Authorization:** bf45c093e542f057caee68c47787e7d6

**More Knowledge on REST API Design**  
Following links will explains you the best practices of REST and other principles.  
1. [RESTful Web services: The basics](https://www.ibm.com/developerworks/webservices/library/ws-restful/)  
2. Stackoverflow [discussion](http://stackoverflow.com/questions/671118/what-exactly-is-restful-programming)  
3. A [video](http://www.youtube.com/watch?v=hdSrT4yjS1g) presentation about REST+JSON API Design – Best Practices for Developers by Les Hazlewood, Stormpath

**2. Prerequisite**

Before diving deep into this article, it is recommended that you have basic knowledge on PHP, MySQL, JSON parsing and Android PHP, MySQL communication. Go through following links to get basic knowledge.

1. [PHP Basics](http://www.w3schools.com/php/)  
2. [MySQL Prepared Statements](http://www.php.net/manual/en/mysqli.quickstart.prepared-statements.php)  
3. [Android JSON Parsing](http://www.androidhive.info/2012/01/android-json-parsing-tutorial/)  
4. [How to connect Android with PHP, MySQL](http://www.androidhive.info/2012/05/how-to-connect-android-with-php-mysql/)

**3. Slim PHP Micro Framework**

Instead of start developing a fresh REST framework from scratch, it is better go with a already proven framework. Then I came across Slim framework and selected it for the following reasons.

**1**. It is very light weight, clean and a beginner can easily understand the framework.  
**2**. Supports all HTTP methods GET, POST, PUT and DELETE which are necessary for a REST API.  
**3**. More importantly it provides a middle layer architecture which will be useful to filter the requests. In our case we can use it for verifying the API Key.

**Downloading Slim Framework**  
Download the Slim framework from [here](https://github.com/codeguy/Slim) (download the stable release) and keep it aside. We are gonna need this some point later after doing required setup.

**4. Installing WAMP Server (Apache, PHP and MySQL)**

WAMP lets you install **Apache**, **PHP** and **MySQL** with a single installer which reduces burden of installing & configuring them separately. Alternatively you can use [XAMP](http://www.apachefriends.org/en/xampp-windows.html), [LAMP](http://en.kioskea.net/faq/1172-lamp-apache-mysql-php-quick-install-on-linux) (on Linux) and [MAMP](http://mamp.info/en/downloads/) (on MAC). WAMP also provides you **phpmyadmin** to easily interact with MySQL database.

Download & install WAMP from [http://www.wampserver.com/en/](http://www.wampserver.com/en/#download-wrapper). Choose the correct version which suits your operating system (32bit or 64bit). Once you have installed it, open the program from **Start -> All Programs -> Wamp Server -> Start WampServer**.

Open <http://localhost/> and <http://localhost/phpmyadmin/> to verify WAMP is installed successfully or not.

**5. Installing Chrome Advanced REST client extension for Testing**

Chrome Advanced REST client extension provides an easy way to test the REST API. It provides lot of options like adding request headers, adding request parameters, changing HTTP method while hitting an url. Install [Advanced REST client](https://chrome.google.com/webstore/detail/advanced-rest-client/hgmloofddffdnphfgcellkdfbfbjeloo) extension in chrome browser. Once you installed it you can find it in chrome Apps or an icon at the top right corner.

Alternatively if you prefer using firefox, you can go for [Poster](https://addons.mozilla.org/en-US/firefox/addon/poster/) add-on to test the API.

**6. REST API for Task Manager App**

To demonstrate REST API I am considering an example of Task Manager App with very minimal functionalities.  
**1**. User related operations like registration and login  
**2**. Task related operations like creating, reading, updating and deleting task. All task related API calls should include API key in Authorization header field.

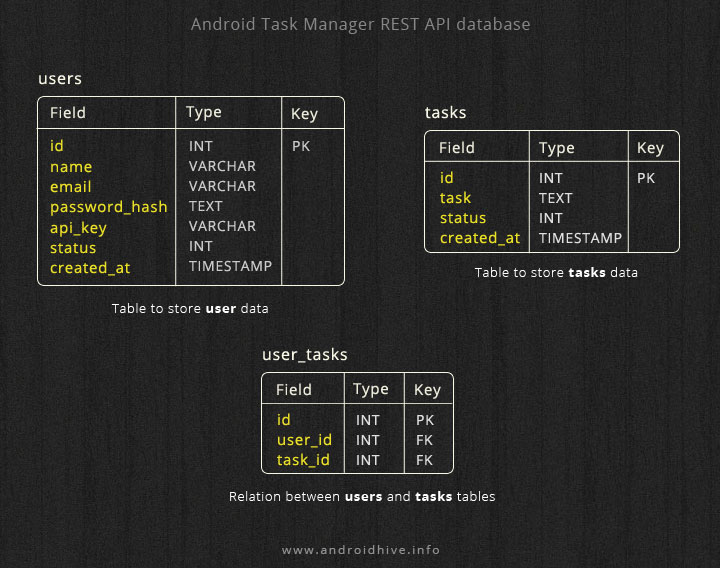
Following are the list of API calls we are going to build in this tutorial. You can notice that same url endpoint is used for multiple api calls, but the difference is the type of HTTP method we use to hit the url. Suppose if we hit **/tasks** with **POST** method, a newer task will be created. As well if we hit **/tasks** with **GET** method, all the tasks will be listed.

**API Url Structure**

|  |  |  |  |
| --- | --- | --- | --- |
| **URL** | **Method** | **Parameters** | **Description** |
| /register | POST | name, email, password | User registration |
| /login | POST | email, password | User login |
| /tasks | POST | task | To create new task |
| /tasks | GET |  | Fetching all tasks |
| /tasks/:id | GET |  | Fetching single task |
| /tasks/:id | PUT |  | Updating single task |
| /tasks/:id | DELETE | task, status | Deleting single task |

**7. Creating MySQL Database**

For this app we don’t need a complex database design. All we need at this stage is only three tables. You can always add few more tables if you want to extend the functionality. I have created three tables **users**, **tasks** and **user\_tasks**.  
*users* – All user related data will be stored here. A row will inserted when a new user register in our app.  
*tasks* – All user tasks data will be stored in this table  
*user\_tasks* – Table used to store the relation between user and his tasks. Basically we store **users id** and **task id** in this table.



Open the phpmyadmin from <http://localhost/phpmyadmin> and execute the following SQL queries. As well if you are familiar with phpmyadmin, you can use phpmyadmin graphical interface to create tables.

|  |
| --- |
| CREATE DATABASE task\_manager;    USE task\_manager;    CREATE TABLE IF NOT EXISTS `users` (    `id` int(11) NOT NULL AUTO\_INCREMENT,    `name` varchar(250) DEFAULT NULL,    `email` varchar(255) NOT NULL,    `password\_hash` text NOT NULL,    `api\_key` varchar(32) NOT NULL,    `status` int(1) NOT NULL DEFAULT '1',    `created\_at` timestamp NOT NULL DEFAULT CURRENT\_TIMESTAMP,    PRIMARY KEY (`id`),    UNIQUE KEY `email` (`email`)  );    CREATE TABLE IF NOT EXISTS `tasks` (    `id` int(11) NOT NULL AUTO\_INCREMENT,    `task` text NOT NULL,    `status` int(1) NOT NULL DEFAULT '0',    `created\_at` timestamp NOT NULL DEFAULT CURRENT\_TIMESTAMP,    PRIMARY KEY (`id`)  );    CREATE TABLE IF NOT EXISTS `user\_tasks` (    `id` int(11) NOT NULL AUTO\_INCREMENT,    `user\_id` int(11) NOT NULL,    `task\_id` int(11) NOT NULL,    PRIMARY KEY (`id`),    KEY `user\_id` (`user\_id`),    KEY `task\_id` (`task\_id`)  );    ALTER TABLE  `user\_tasks` ADD FOREIGN KEY (  `user\_id` ) REFERENCES  `task\_manager`.`users` (  `id`  ) ON DELETE CASCADE ON UPDATE CASCADE ;    ALTER TABLE  `user\_tasks` ADD FOREIGN KEY (  `task\_id` ) REFERENCES  `task\_manager`.`tasks` (  `id`  ) ON DELETE CASCADE ON UPDATE CASCADE ; |

After executing these queries go through each tables and make sure that everything created correctly.

**How to create REST API for Android app using PHP, Slim and MySQL – Day 2/2**

by [Ravi Tamada](http://www.androidhive.info/author/admin/)/ January 20, 2014/ [**483** Comments](http://www.androidhive.info/2014/01/how-to-create-rest-api-for-android-app-using-php-slim-and-mysql-day-23/#disqus_thread)

The previous day [How to create REST API for Android app using PHP, Slim and MySQL – Day ½](http://www.androidhive.info/2014/01/how-to-create-rest-api-for-android-app-using-php-slim-and-mysql-day-12-2/), we have learned fundamental concepts about REST API and preparing your development environment ready by installing the required tools needed. I hope everyone got good knowledge about REST and other technical areas. Also I am assuming that you got all the required tools installed.

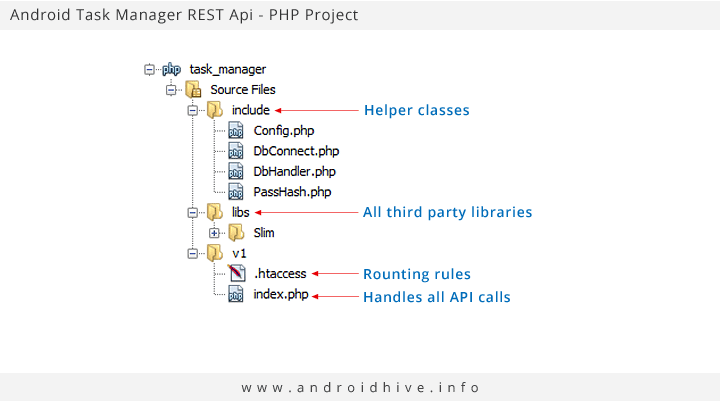
Today we are going to learn how to setup a PHP project and writing the actual code for REST API. Also we’ll learn writing necessary SQL queries to perform database CRUD operations.

[DOWNLOAD CODE](http://download.androidhive.info/download?code=Af5zV3Fmd8DKUL4fkN66pbPXsEO5oiaWabC%2BZ0Ekem3cdd48fb1710IE%2B80r2vLVAFkr1HX7preT9Q0W4toErT3nC5MwXOW3vIXD%2BelsMUltp0t9Bm%2ByFvKqA%3D%3DzxEDnj3DwtbbqLmQ484q9suQqfzVv6OacCe)

**8. Starting PHP Project**

As we all know IDEs make development process easier. So I recommend you use an IDE for developing the PHP project instead of using plain notepad. You can go for [Eclipse](http://www.eclipse.org/downloads/packages/eclipse-php-developers/heliosr), [Aptana Studio](http://www.aptana.com/products/php/), [PhpStorm](http://www.jetbrains.com/phpstorm/download/) or [Netbeans](https://netbeans.org/downloads/). But I personally felt very comfortable using Netbeans for PHP projects.

**PHP Project directory structure**  
The following diagram will give you an idea about the directory structure of the project which we are going to develop now.



*libs* – All the third party libraries goes here. In our case we place Slim library here  
*include* – All the helpers classes we build placed here  
*index.php* – Takes care of all the API requests  
*.htaccess* – Rules for url structure and other apache rules

Now let’s start the PHP project

**1**. Go to the directory where WAMP is installed. In general wamp will be installed in **C:\wamp**. (If you have installed any other software rather than WAMP, you should go to the directory recommended by that software).

**2**. As a first step we start with creating required directories. Inside **wamp** folder go to **www** folder **(c:\wamp\www\)** and create a folder named **task\_manager**. This folder will be the parent directory of our project. Inside task\_manager create two more folders named **libs**, **include** and **v1**.

**3**. Now the paste the **Slim library** inside **libs** folder. The download link for Slim is provided in previous part.

**4**. Normally Slim framework works when **index.php** includes in the url which makes url not well-formed. So using the **.htacess** rules we can get rid of index.php from the url and make some friendly urls. Inside **v1** folder create a file named **.htaccess** and paste the following code. (Note that this file name shouldn’t include any additional extension in the name like .txt)

|  |
| --- |
| .htaccess |
| RewriteEngine On  RewriteCond %{REQUEST\_FILENAME} !-f  RewriteRule ^(.\*)$ %{ENV:BASE}index.php [QSA,L] |

**8.1 Preparing Helper Classes**

First we start writing set of helper classes required in this project. These helper classes provides necessary functions required to interact with the database.

**5**. Inside **include** folder create file named **Config.php** with following content. This file contains the entire project configuration like database connection parameters and other variables.

|  |
| --- |
| Config.php |
| <?php  /\*\*   \* Database configuration   \*/  define('DB\_USERNAME', 'root');  define('DB\_PASSWORD', '');  define('DB\_HOST', 'localhost');  define('DB\_NAME', 'task\_manager');    define('USER\_CREATED\_SUCCESSFULLY', 0);  define('USER\_CREATE\_FAILED', 1);  define('USER\_ALREADY\_EXISTED', 2);  ?> |

**6**. Create another class named **DbConnect.php** This class file mainly takes care of database connection.

|  |
| --- |
| DbConnect.php |
| <?php    /\*\*   \* Handling database connection   \*   \* @author Ravi Tamada   \*/  class DbConnect {        private $conn;        function \_\_construct() {      }        /\*\*       \* Establishing database connection       \* @return database connection handler       \*/      function connect() {          include\_once dirname(\_\_FILE\_\_) . './Config.php';            // Connecting to mysql database          $this->conn = new mysqli(DB\_HOST, DB\_USERNAME, DB\_PASSWORD, DB\_NAME);            // Check for database connection error          if (mysqli\_connect\_errno()) {              echo "Failed to connect to MySQL: " . mysqli\_connect\_error();          }            // returing connection resource          return $this->conn;      }    }    ?> |

**Encrypting the password**  
**7**. The best way to secure the user passwords is not store them as plain text, instead all the passwords should be encrypted before storing in db. The following class takes care of encrypting the user password. Create another file named **PassHash.php** and paste the following code.

|  |
| --- |
| PassHash.php |
| <?php    class PassHash {        // blowfish      private static $algo = '$2a';      // cost parameter      private static $cost = '$10';        // mainly for internal use      public static function unique\_salt() {          return substr(sha1(mt\_rand()), 0, 22);      }        // this will be used to generate a hash      public static function hash($password) {            return crypt($password, self::$algo .                  self::$cost .                  '$' . self::unique\_salt());      }        // this will be used to compare a password against a hash      public static function check\_password($hash, $password) {          $full\_salt = substr($hash, 0, 29);          $new\_hash = crypt($password, $full\_salt);          return ($hash == $new\_hash);      }    }    ?> |

**8**. Now create another class named **DbHandler.php** This class is one of the important files in our project which provides necessary functions to perform CRUD operations on the database. Every function is self explanatory by it’s name and comments, I don’t have to have to explain much about them.

|  |
| --- |
| DbHandler.php |
| <?php    /\*\*   \* Class to handle all db operations   \* This class will have CRUD methods for database tables   \*   \* @author Ravi Tamada   \*/  class DbHandler {        private $conn;        function \_\_construct() {          require\_once dirname(\_\_FILE\_\_) . './DbConnect.php';          // opening db connection          $db = new DbConnect();          $this->conn = $db->connect();      }        /\* ------------- `users` table method ------------------ \*/        /\*\*       \* Creating new user       \* @param String $name User full name       \* @param String $email User login email id       \* @param String $password User login password       \*/      public function createUser($name, $email, $password) {          require\_once 'PassHash.php';          $response = array();            // First check if user already existed in db          if (!$this->isUserExists($email)) {              // Generating password hash              $password\_hash = PassHash::hash($password);                // Generating API key              $api\_key = $this->generateApiKey();                // insert query              $stmt = $this->conn->prepare("INSERT INTO users(name, email, password\_hash, api\_key, status) values(?, ?, ?, ?, 1)");              $stmt->bind\_param("ssss", $name, $email, $password\_hash, $api\_key);                $result = $stmt->execute();                $stmt->close();                // Check for successful insertion              if ($result) {                  // User successfully inserted                  return USER\_CREATED\_SUCCESSFULLY;              } else {                  // Failed to create user                  return USER\_CREATE\_FAILED;              }          } else {              // User with same email already existed in the db              return USER\_ALREADY\_EXISTED;          }            return $response;      }        /\*\*       \* Checking user login       \* @param String $email User login email id       \* @param String $password User login password       \* @return boolean User login status success/fail       \*/      public function checkLogin($email, $password) {          // fetching user by email          $stmt = $this->conn->prepare("SELECT password\_hash FROM users WHERE email = ?");            $stmt->bind\_param("s", $email);            $stmt->execute();            $stmt->bind\_result($password\_hash);            $stmt->store\_result();            if ($stmt->num\_rows > 0) {              // Found user with the email              // Now verify the password                $stmt->fetch();                $stmt->close();                if (PassHash::check\_password($password\_hash, $password)) {                  // User password is correct                  return TRUE;              } else {                  // user password is incorrect                  return FALSE;              }          } else {              $stmt->close();                // user not existed with the email              return FALSE;          }      }        /\*\*       \* Checking for duplicate user by email address       \* @param String $email email to check in db       \* @return boolean       \*/      private function isUserExists($email) {          $stmt = $this->conn->prepare("SELECT id from users WHERE email = ?");          $stmt->bind\_param("s", $email);          $stmt->execute();          $stmt->store\_result();          $num\_rows = $stmt->num\_rows;          $stmt->close();          return $num\_rows > 0;      }        /\*\*       \* Fetching user by email       \* @param String $email User email id       \*/      public function getUserByEmail($email) {          $stmt = $this->conn->prepare("SELECT name, email, api\_key, status, created\_at FROM users WHERE email = ?");          $stmt->bind\_param("s", $email);          if ($stmt->execute()) {              $user = $stmt->get\_result()->fetch\_assoc();              $stmt->close();              return $user;          } else {              return NULL;          }      }        /\*\*       \* Fetching user api key       \* @param String $user\_id user id primary key in user table       \*/      public function getApiKeyById($user\_id) {          $stmt = $this->conn->prepare("SELECT api\_key FROM users WHERE id = ?");          $stmt->bind\_param("i", $user\_id);          if ($stmt->execute()) {              $api\_key = $stmt->get\_result()->fetch\_assoc();              $stmt->close();              return $api\_key;          } else {              return NULL;          }      }        /\*\*       \* Fetching user id by api key       \* @param String $api\_key user api key       \*/      public function getUserId($api\_key) {          $stmt = $this->conn->prepare("SELECT id FROM users WHERE api\_key = ?");          $stmt->bind\_param("s", $api\_key);          if ($stmt->execute()) {              $user\_id = $stmt->get\_result()->fetch\_assoc();              $stmt->close();              return $user\_id;          } else {              return NULL;          }      }        /\*\*       \* Validating user api key       \* If the api key is there in db, it is a valid key       \* @param String $api\_key user api key       \* @return boolean       \*/      public function isValidApiKey($api\_key) {          $stmt = $this->conn->prepare("SELECT id from users WHERE api\_key = ?");          $stmt->bind\_param("s", $api\_key);          $stmt->execute();          $stmt->store\_result();          $num\_rows = $stmt->num\_rows;          $stmt->close();          return $num\_rows > 0;      }        /\*\*       \* Generating random Unique MD5 String for user Api key       \*/      private function generateApiKey() {          return md5(uniqid(rand(), true));      }        /\* ------------- `tasks` table method ------------------ \*/        /\*\*       \* Creating new task       \* @param String $user\_id user id to whom task belongs to       \* @param String $task task text       \*/      public function createTask($user\_id, $task) {          $stmt = $this->conn->prepare("INSERT INTO tasks(task) VALUES(?)");          $stmt->bind\_param("s", $task);          $result = $stmt->execute();          $stmt->close();            if ($result) {              // task row created              // now assign the task to user              $new\_task\_id = $this->conn->insert\_id;              $res = $this->createUserTask($user\_id, $new\_task\_id);              if ($res) {                  // task created successfully                  return $new\_task\_id;              } else {                  // task failed to create                  return NULL;              }          } else {              // task failed to create              return NULL;          }      }        /\*\*       \* Fetching single task       \* @param String $task\_id id of the task       \*/      public function getTask($task\_id, $user\_id) {          $stmt = $this->conn->prepare("SELECT t.id, t.task, t.status, t.created\_at from tasks t, user\_tasks ut WHERE t.id = ? AND ut.task\_id = t.id AND ut.user\_id = ?");          $stmt->bind\_param("ii", $task\_id, $user\_id);          if ($stmt->execute()) {              $task = $stmt->get\_result()->fetch\_assoc();              $stmt->close();              return $task;          } else {              return NULL;          }      }        /\*\*       \* Fetching all user tasks       \* @param String $user\_id id of the user       \*/      public function getAllUserTasks($user\_id) {          $stmt = $this->conn->prepare("SELECT t.\* FROM tasks t, user\_tasks ut WHERE t.id = ut.task\_id AND ut.user\_id = ?");          $stmt->bind\_param("i", $user\_id);          $stmt->execute();          $tasks = $stmt->get\_result();          $stmt->close();          return $tasks;      }        /\*\*       \* Updating task       \* @param String $task\_id id of the task       \* @param String $task task text       \* @param String $status task status       \*/      public function updateTask($user\_id, $task\_id, $task, $status) {          $stmt = $this->conn->prepare("UPDATE tasks t, user\_tasks ut set t.task = ?, t.status = ? WHERE t.id = ? AND t.id = ut.task\_id AND ut.user\_id = ?");          $stmt->bind\_param("siii", $task, $status, $task\_id, $user\_id);          $stmt->execute();          $num\_affected\_rows = $stmt->affected\_rows;          $stmt->close();          return $num\_affected\_rows > 0;      }        /\*\*       \* Deleting a task       \* @param String $task\_id id of the task to delete       \*/      public function deleteTask($user\_id, $task\_id) {          $stmt = $this->conn->prepare("DELETE t FROM tasks t, user\_tasks ut WHERE t.id = ? AND ut.task\_id = t.id AND ut.user\_id = ?");          $stmt->bind\_param("ii", $task\_id, $user\_id);          $stmt->execute();          $num\_affected\_rows = $stmt->affected\_rows;          $stmt->close();          return $num\_affected\_rows > 0;      }        /\* ------------- `user\_tasks` table method ------------------ \*/        /\*\*       \* Function to assign a task to user       \* @param String $user\_id id of the user       \* @param String $task\_id id of the task       \*/      public function createUserTask($user\_id, $task\_id) {          $stmt = $this->conn->prepare("INSERT INTO user\_tasks(user\_id, task\_id) values(?, ?)");          $stmt->bind\_param("ii", $user\_id, $task\_id);          $result = $stmt->execute();          $stmt->close();          return $result;      }    }    ?> |

**8.2 Handling the API calls**

Now we have all the required classes for the REST API. Now we can start the code to handle all individual api calls.

**8**. Inside **v1** folder create a file named **index.php** and add the following code. Here we are including required libraries and other helper functions.

**verifyRequiredParams()** – This function verifies the mandatory parameters in the request.  
**validateEmail()** – Verifies whether email address is valid one or not.  
**echoRespnse()** – This function will echo the JSON response with a status code.

|  |
| --- |
| index.php |
| <?php    require\_once '../include/DbHandler.php';  require\_once '../include/PassHash.php';  require '.././libs/Slim/Slim.php';    \Slim\Slim::registerAutoloader();    $app = new \Slim\Slim();    // User id from db - Global Variable  $user\_id = NULL;    /\*\*   \* Verifying required params posted or not   \*/  function verifyRequiredParams($required\_fields) {      $error = false;      $error\_fields = "";      $request\_params = array();      $request\_params = $\_REQUEST;      // Handling PUT request params      if ($\_SERVER['REQUEST\_METHOD'] == 'PUT') {          $app = \Slim\Slim::getInstance();          parse\_str($app->request()->getBody(), $request\_params);      }      foreach ($required\_fields as $field) {          if (!isset($request\_params[$field]) || strlen(trim($request\_params[$field])) <= 0) {              $error = true;              $error\_fields .= $field . ', ';          }      }        if ($error) {          // Required field(s) are missing or empty          // echo error json and stop the app          $response = array();          $app = \Slim\Slim::getInstance();          $response["error"] = true;          $response["message"] = 'Required field(s) ' . substr($error\_fields, 0, -2) . ' is missing or empty';          echoRespnse(400, $response);          $app->stop();      }  }    /\*\*   \* Validating email address   \*/  function validateEmail($email) {      $app = \Slim\Slim::getInstance();      if (!filter\_var($email, FILTER\_VALIDATE\_EMAIL)) {          $response["error"] = true;          $response["message"] = 'Email address is not valid';          echoRespnse(400, $response);          $app->stop();      }  }    /\*\*   \* Echoing json response to client   \* @param String $status\_code Http response code   \* @param Int $response Json response   \*/  function echoRespnse($status\_code, $response) {      $app = \Slim\Slim::getInstance();      // Http response code      $app->status($status\_code);        // setting response content type to json      $app->contentType('application/json');        echo json\_encode($response);  }    $app->run();  ?> |

**The JSON response**

On calling every API request a JSON response will be issued with a HTTP status code. On the client side you have to verify the response http status code. If the status is **200**, the request is processed successfully. Also you can notice a **“error”** node in the response. If the error value is **true**, that means some error occurred while processing the user data.

**Api Calls without Authentication (without API key in the request header)**

These calls don’t have to include Api Key in the request header. The main purpose of these calls is to interact with database without any authentication. User **registration** and **login** comes under this category.

**⇒ User Registration**

In order to interact with the API, the user has to register in our system first. Once he registered an API key will be generated and stored in the database. This API key will be private to that user only.

**9**. Add the following code in **index.php**. This function handles user registration.

|  |
| --- |
| index.php |
| /\*\*   \* User Registration   \* url - /register   \* method - POST   \* params - name, email, password   \*/  $app->post('/register', function() use ($app) {              // check for required params              verifyRequiredParams(array('name', 'email', 'password'));                $response = array();                // reading post params              $name = $app->request->post('name');              $email = $app->request->post('email');              $password = $app->request->post('password');                // validating email address              validateEmail($email);                $db = new DbHandler();              $res = $db->createUser($name, $email, $password);                if ($res == USER\_CREATED\_SUCCESSFULLY) {                  $response["error"] = false;                  $response["message"] = "You are successfully registered";                  echoRespnse(201, $response);              } else if ($res == USER\_CREATE\_FAILED) {                  $response["error"] = true;                  $response["message"] = "Oops! An error occurred while registereing";                  echoRespnse(200, $response);              } else if ($res == USER\_ALREADY\_EXISTED) {                  $response["error"] = true;                  $response["message"] = "Sorry, this email already existed";                  echoRespnse(200, $response);              }          }); |

In the following table you can find the API request information about the URL, HTTP method and the parameters needed to be posted.

|  |  |
| --- | --- |
| **URL** | /register |
| **Method** | POST |
| **Params** | name, email, password |

Upon the successful registration the following json response will be issued.

|  |
| --- |
| {      "error": false,      "message": "You are successfully registered"  } |

If the request is missing mandatory parameters the following json will be issued.

|  |
| --- |
| {      "error": true,      "message": "Required field(s) email, password is missing or empty"  } |

**⇒ User Login**

**10**. Add the following code to handle user login. After verifying user credentials, the API Key for that user will be issued in the json response. The api key should be included in the request header in all remaining api calls.

|  |  |
| --- | --- |
| index.php | |
| /\*\*   \* User Login   \* url - /login   \* method - POST   \* params - email, password   \*/  $app->post('/login', function() use ($app) {              // check for required params              verifyRequiredParams(array('email', 'password'));                // reading post params              $email = $app->request()->post('email');              $password = $app->request()->post('password');              $response = array();                $db = new DbHandler();              // check for correct email and password              if ($db->checkLogin($email, $password)) {                  // get the user by email                  $user = $db->getUserByEmail($email);                    if ($user != NULL) {                      $response["error"] = false;                      $response['name'] = $user['name'];                      $response['email'] = $user['email'];                      $response['apiKey'] = $user['api\_key'];                      $response['createdAt'] = $user['created\_at'];                  } else {                      // unknown error occurred                      $response['error'] = true;                      $response['message'] = "An error occurred. Please try again";                  }              } else {                  // user credentials are wrong                  $response['error'] = true;                  $response['message'] = 'Login failed. Incorrect credentials';              }                echoRespnse(200, $response);          }); | |
| **URL** | /login |
| **Method** | POST |
| **Params** | email, password |

On successful login the following json will be issued.

|  |
| --- |
| {      "error": false,      "name": "Ravi Tamada",      "email": "ravi@gmail.com",      "apiKey": "940bb12af8d7b040876f60f965c5be6d",      "createdAt": "2014-01-07 23:38:35"  } |

If the credentials are wrong, you can expect the following json.

|  |
| --- |
| {      "error": true,      "message": "Login failed. Incorrect credentials"  } |

**⇒ Verifying API Key**

While dealing with task data, we need to identify the user using the API key in the request header by reading **Authorization** field. Basically we’ll look into database for matched API key and get the appropriate user. If the API key not present in users table, then we’ll stop the execution and echo the error json.

**11**. Add the following method in **index.php**. The method **authenticate()** will be executed every time before doing any task related operations on database.

|  |
| --- |
| index.php |
| /\*\*   \* Adding Middle Layer to authenticate every request   \* Checking if the request has valid api key in the 'Authorization' header   \*/  function authenticate(\Slim\Route $route) {      // Getting request headers      $headers = apache\_request\_headers();      $response = array();      $app = \Slim\Slim::getInstance();        // Verifying Authorization Header      if (isset($headers['Authorization'])) {          $db = new DbHandler();            // get the api key          $api\_key = $headers['Authorization'];          // validating api key          if (!$db->isValidApiKey($api\_key)) {              // api key is not present in users table              $response["error"] = true;              $response["message"] = "Access Denied. Invalid Api key";              echoRespnse(401, $response);              $app->stop();          } else {              global $user\_id;              // get user primary key id              $user = $db->getUserId($api\_key);              if ($user != NULL)                  $user\_id = $user["id"];          }      } else {          // api key is missing in header          $response["error"] = true;          $response["message"] = "Api key is misssing";          echoRespnse(400, $response);          $app->stop();      }  } |

If the api key is missing in the request header, the following json will be echoed with 400 status code.

|  |
| --- |
| {      "error": true,      "message": "Api key is misssing"  } |

If the api key is not valid following json will echoed with 401 status code.

|  |
| --- |
| {      "error": true,      "message": "Access Denied. Invalid Api key"  } |

**Api Calls with Authentication (Including API key in the request)**   
Following are the API calls should have an Api Key in the request header. These api calls primarily deals the user’s task data like creating, reading, updating and deleting.

**⇒ Creating New Task**

**12**. Add the follwing method to create a new task. Here you can notice that **authenticate** method is called to verify the Api key before inserting a new task.

|  |  |
| --- | --- |
| index.php | |
| /\*\*   \* Creating new task in db   \* method POST   \* params - name   \* url - /tasks/   \*/  $app->post('/tasks', 'authenticate', function() use ($app) {              // check for required params              verifyRequiredParams(array('task'));                $response = array();              $task = $app->request->post('task');                global $user\_id;              $db = new DbHandler();                // creating new task              $task\_id = $db->createTask($user\_id, $task);                if ($task\_id != NULL) {                  $response["error"] = false;                  $response["message"] = "Task created successfully";                  $response["task\_id"] = $task\_id;              } else {                  $response["error"] = true;                  $response["message"] = "Failed to create task. Please try again";              }              echoRespnse(201, $response);          }); | |
| **URL** | /tasks |
| **Method** | POST |
| **Params** | task |

On successful creation of new task following json will be issued. If you got this json, you can see new row inserted in **tasks** and **user\_tasks** tables.

|  |
| --- |
| {      "error": false,      "message": "Task created successfully",      "task\_id": 1  } |

**⇒ Getting All Tasks**

**13**. Following method will list down all user’s tasks. We don’t have to submit any params for this api call.

|  |  |
| --- | --- |
| index.php | |
| /\*\*   \* Listing all tasks of particual user   \* method GET   \* url /tasks   \*/  $app->get('/tasks', 'authenticate', function() {              global $user\_id;              $response = array();              $db = new DbHandler();                // fetching all user tasks              $result = $db->getAllUserTasks($user\_id);                $response["error"] = false;              $response["tasks"] = array();                // looping through result and preparing tasks array              while ($task = $result->fetch\_assoc()) {                  $tmp = array();                  $tmp["id"] = $task["id"];                  $tmp["task"] = $task["task"];                  $tmp["status"] = $task["status"];                  $tmp["createdAt"] = $task["created\_at"];                  array\_push($response["tasks"], $tmp);              }                echoRespnse(200, $response);          }); | |
| **URL** | /tasks |
| **Method** | GET |
| **Params** | – |

Following json will be issued for list of tasks. The **“tasks”** represents list of tasks as an array. Also if the **“status”** is **0**, that means the task is not done yet.

|  |
| --- |
| {      "error": false,      "tasks": [          {              "id": 1,              "task": "Complete REST article by Sunday",              "status": 0,              "createdAt": "2014-01-08 23:35:45"          },          {              "id": 2,              "task": "Book bus tickets!",              "status": 0,              "createdAt": "2014-01-08 23:56:52"          }      ]  } |

**⇒ Getting Single Task**

**14**. Following method will fetch details of single task. You need to append the **task id** with a **/** to url. For an example if you want details of **task 15**, the url will be **/tasks/15**.

|  |  |
| --- | --- |
| index.php | |
| /\*\*   \* Listing single task of particual user   \* method GET   \* url /tasks/:id   \* Will return 404 if the task doesn't belongs to user   \*/  $app->get('/tasks/:id', 'authenticate', function($task\_id) {              global $user\_id;              $response = array();              $db = new DbHandler();                // fetch task              $result = $db->getTask($task\_id, $user\_id);                if ($result != NULL) {                  $response["error"] = false;                  $response["id"] = $result["id"];                  $response["task"] = $result["task"];                  $response["status"] = $result["status"];                  $response["createdAt"] = $result["created\_at"];                  echoRespnse(200, $response);              } else {                  $response["error"] = true;                  $response["message"] = "The requested resource doesn't exists";                  echoRespnse(404, $response);              }          }); | |
| **URL** | /tasks/id (id should be replaced with task id) |
| **Method** | GET |
| **Params** | – |

The details of a single task will be in following json format.

|  |
| --- |
| {      "error": false,      "id": 2,      "task": "Book bus tickets!",      "status": 0,      "createdAt": "2014-01-08 23:56:52"  } |

If you pass a task id which is not there in the database, you will get **404** not found error.

**⇒ Updating Task**

**15**. Following code will take care of updating a task. The url for this api call is same as getting the details of single task, only difference is we should use **PUT** method instead of GET.

|  |  |
| --- | --- |
| index.php | |
| /\*\*   \* Updating existing task   \* method PUT   \* params task, status   \* url - /tasks/:id   \*/  $app->put('/tasks/:id', 'authenticate', function($task\_id) use($app) {              // check for required params              verifyRequiredParams(array('task', 'status'));                global $user\_id;              $task = $app->request->put('task');              $status = $app->request->put('status');                $db = new DbHandler();              $response = array();                // updating task              $result = $db->updateTask($user\_id, $task\_id, $task, $status);              if ($result) {                  // task updated successfully                  $response["error"] = false;                  $response["message"] = "Task updated successfully";              } else {                  // task failed to update                  $response["error"] = true;                  $response["message"] = "Task failed to update. Please try again!";              }              echoRespnse(200, $response);          }); | |
| **URL** | /tasks/id (id should be replaced with task id) |
| **Method** | PUT |
| **Params** | task, status (0 or 1) |

Upon successful updation you will get following json.

|  |
| --- |
| {      "error": false,      "message": "Task updated successfully"  } |

**⇒ Deleting Task**

**16**. Again delete task url is same as update task, but this requires **DELETE** method.

|  |  |
| --- | --- |
| index.php | |
| /\*\*   \* Deleting task. Users can delete only their tasks   \* method DELETE   \* url /tasks   \*/  $app->delete('/tasks/:id', 'authenticate', function($task\_id) use($app) {              global $user\_id;                $db = new DbHandler();              $response = array();              $result = $db->deleteTask($user\_id, $task\_id);              if ($result) {                  // task deleted successfully                  $response["error"] = false;                  $response["message"] = "Task deleted succesfully";              } else {                  // task failed to delete                  $response["error"] = true;                  $response["message"] = "Task failed to delete. Please try again!";              }              echoRespnse(200, $response);          }); | |
| **URL** | /tasks/id (id should be replaced with task id) |
| **Method** | DELETE |
| **Params** | – |

You will get following json if the task is deleted successfully.

|  |
| --- |
| {      "error": false,      "message": "Task deleted succesfully"  } |

Here we completes the PHP and MySQL part. Now it’s time to move on to testing the API just to make sure that whatever code we have written is working.

**Testing the API**

Following is the list of URL we need to test using Chrome Advanced REST client extension with possible combinations of inputs.

|  |  |  |  |
| --- | --- | --- | --- |
| **URL** | **Method** | **Parameters** | **Description** |
| http://localhost/task\_manager/v1/register | POST | name, email, password | User registration |
| http://localhost/task\_manager/v1/login | POST | email, password | User login |
| http://localhost/task\_manager/v1/tasks | POST | task | To create new task |
| http://localhost/task\_manager/v1/tasks | GET |  | Fetching all tasks |
| http://localhost/task\_manager/v1/tasks/:id | GET |  | Fetching single task |
| http://localhost/task\_manager/v1/tasks/:id | PUT |  | Updating single task |
| http://localhost/task\_manager/v1/tasks/:id | DELETE | task, status | Deleting single task |

The following video shows you how to test the API thoroughly.

REST API Testing

**What’s Next?**

Until now we have done a great job by developing fully functional REST services from scratch. But the services are limited to localhost. In other words your app can’t consume them if you want your app to be on playstore. So to make the services globally accessible, we need to host them on a public server. In the next part [Android Hosting PHP, MySQL RESTful services to DigitalOcean](http://www.androidhive.info/2015/03/android-hosting-php-mysql-restful-services-to-digitalocean/), you will learn the complete process of buying the hosting space, deploying the services and finally mapping to a real domain URL.