Mobile Applications for Sensing and Control

Sep Makhsous

Week 7

Announcements

Introduction to API Calls

- What is an API? - APIs (Application Programming Interfaces) allow different software systems to communicate, acting as a contract where requests and responses are predefined.
- Importance of API calls in mobile applications - APIs enable dynamic features by accessing web services like Google Maps or weather data, essential for modern mobile apps.

How do APIs work?

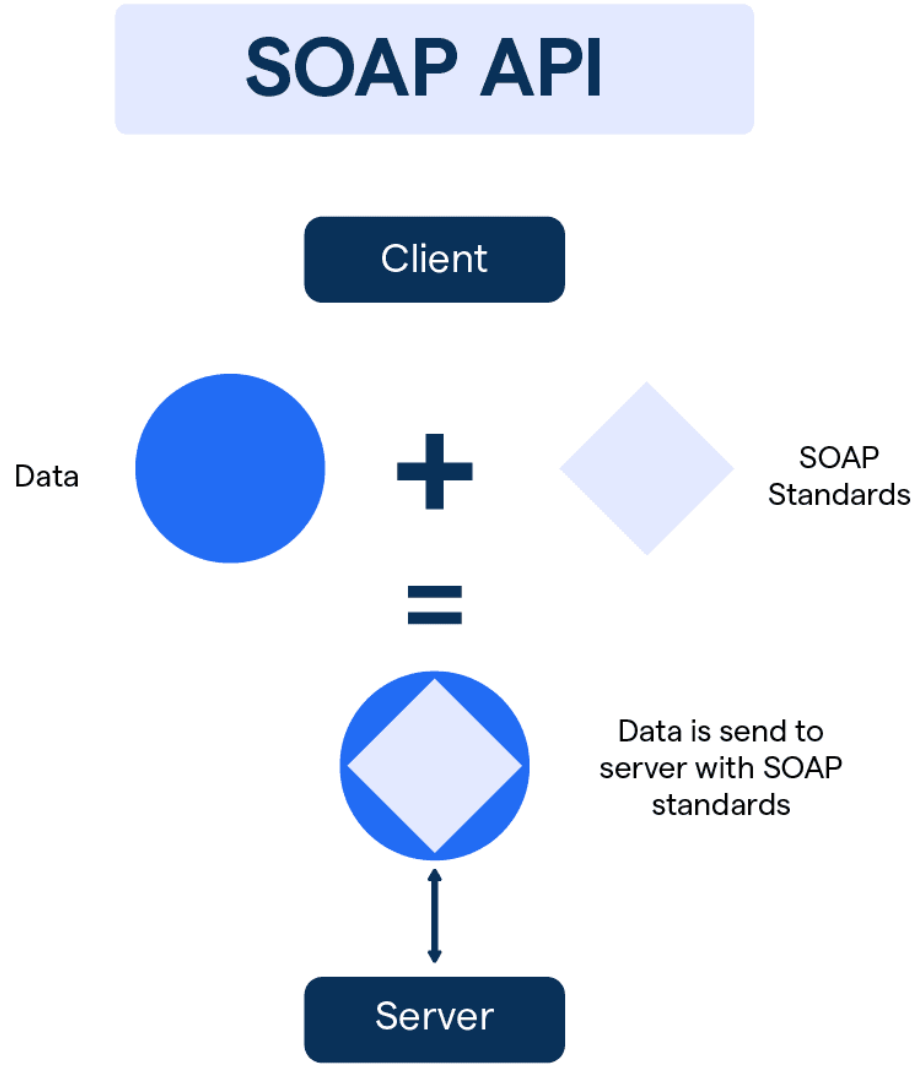
- API architecture is usually explained in terms of client and server. The application sending the request is called the client, and the application sending the response is called the server. So in the weather example, the bureau's weather database is the server, and the mobile app is the client.

How do APIs work?

- There are four different ways that APIs can work depending on when and why they were created.
  - SOAP APIs
  - RPC APIs
  - WebSocket APIs
  - REST APIs

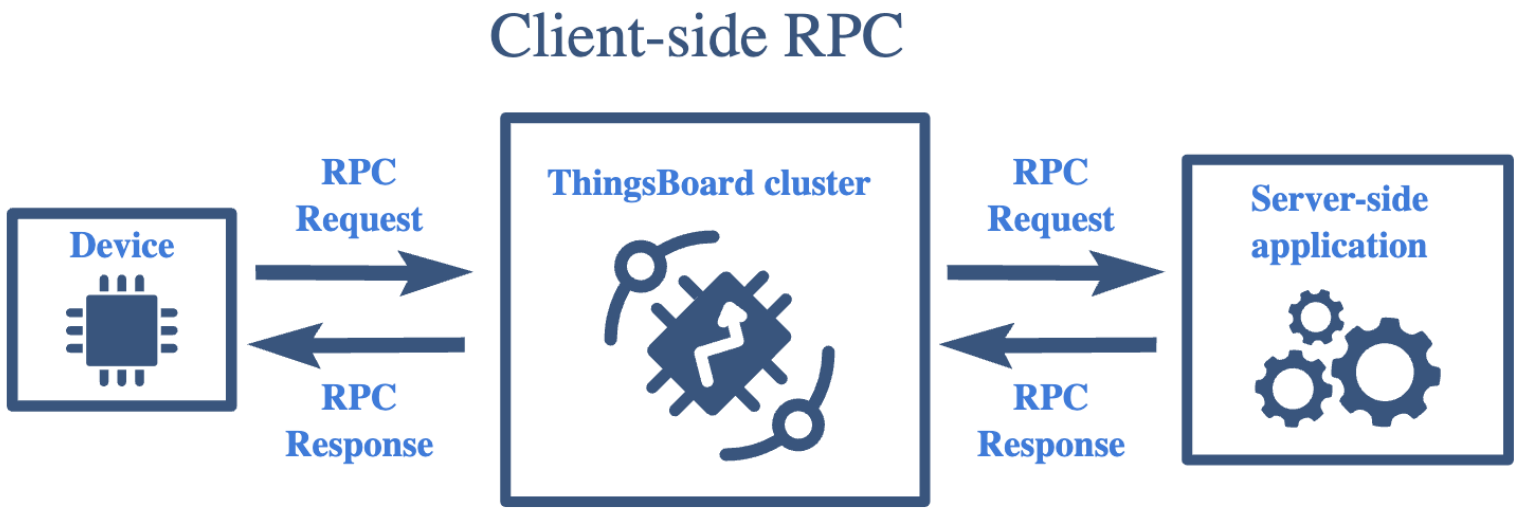
SOAP

- These APIs use Simple Object Access Protocols. Client and server exchange messages using XML. This is a less flexible API that was more popular in the past.



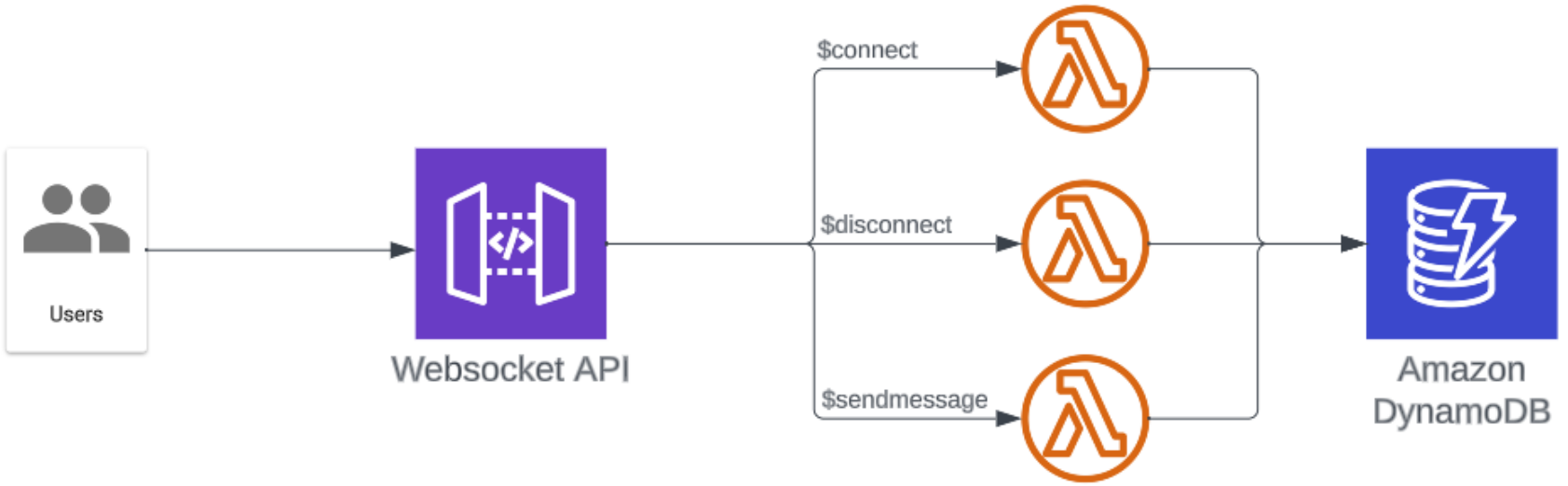
RPC

- These APIs are called Remote Procedure Calls. The client completes a function (or procedure) on the server, and the server sends the output back to the client.



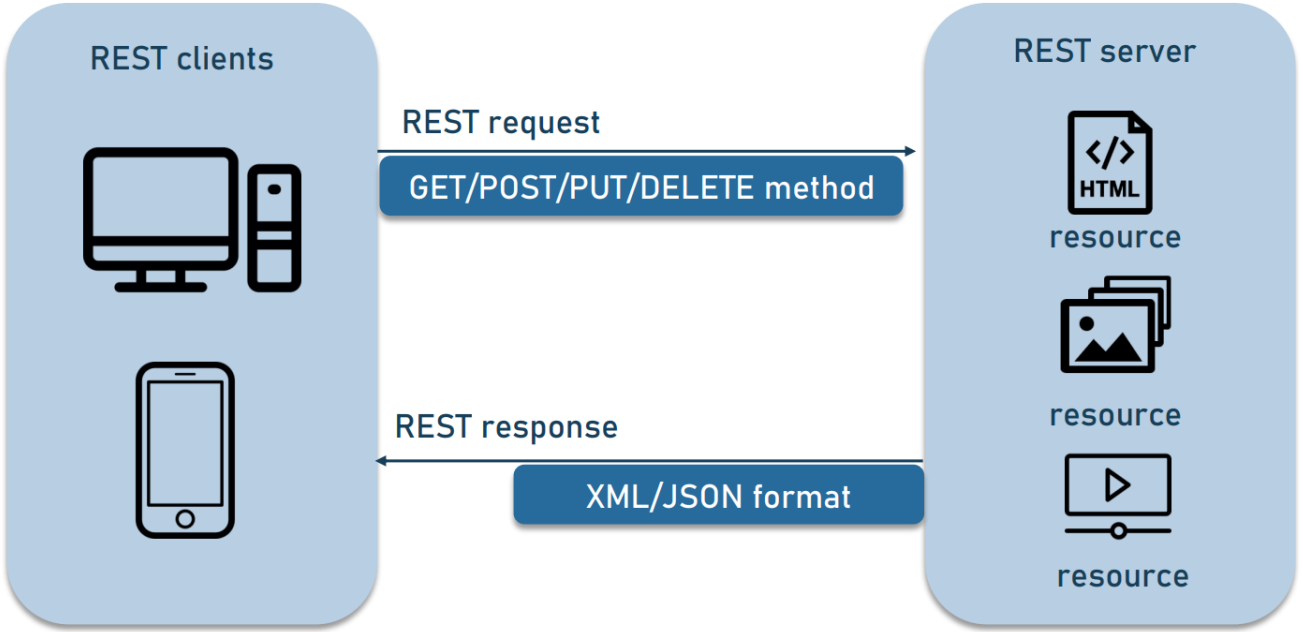
WebSocket

- Websocket API is another modern web API development that uses JSON objects to pass data. A WebSocket API supports two-way communication between client apps and the server. The server can send callback messages to connected clients, making it more efficient than REST API.



REST

- These are the most popular and flexible APIs found on the web today. The client sends requests to the server as data. The server uses this client input to start internal functions and returns output data back to the client.



How to use an API?

The steps to implement a new API include:

- Obtaining an API key. This is done by creating a verified account with the API provider.
- Set up an HTTP API client. This tool allows you to structure API requests easily using the API keys received.
- If you don’t have an API client, you can try to structure the request yourself in your browser by referring to the API documentation.
- Once you are comfortable with the new API syntax, you can start using it in your code.

Setting Up the Request

- Choosing the HTTP method: Depending on the action (retrieve, create, update, delete), the method could be GET, POST, PUT, DELETE, etc.
- Constructing the URL: This includes the base URL, the specific endpoint, and any necessary query parameters.
- Adding headers: Common headers include content type (e.g., application/json), authorization tokens, and other API-specific needs.
- Body data (if necessary): For methods like POST or PUT, the request may need to include data (payload) in the body. This data should be formatted as specified by the API (often as a JSON object).

HTTP Methods

- GET
  - Purpose: Used to retrieve data from a server at the specified resource.
  - Characteristics: GET requests should only retrieve data and have no other effect on the data.
  - Data Sending: Sends data in the URL, typically through query parameters.
- POST
  - Purpose: Used to send data to a server to create a new resource.
  - Characteristics: POST requests are often used when the data is too large or too complex to be sent as part of the URL. It can also be used to submit forms.
  - Data Sending: Sends data in the body of the request, not in the URL.
- PUT
  - Purpose: Used to send data to a server to update or replace an existing resource.
  - Characteristics: PUT requests replace the entire resource if it exists, otherwise create a new resource.
  - Data Sending: Sends data in the body of the request, similar to POST.
- DELETE
  - Purpose: Used to remove a resource specified by a URI.
  - Characteristics: Upon successful deletion, it usually returns an HTTP success status.

Example Scenario

- Suppose we have an API that manages a simple database of users, hosted at https://api.example.com/users.
- Here's how you might use these methods:

GET https://api.example.com/users // Retrieves list of all users

GET https://api.example.com/users/123 // Retrieves details of user with ID 123

Example Scenario

POST https://api.example.com/users

Content-Type: application/json

```
{
  "name": "Jane Doe",
  "email": "jane.doe@example.com"
}
```

Example Scenario

PUT https://api.example.com/users/123

Content-Type: application/json

```
{
  "name": "Jane Roe",
  "email": "jane.roe@example.com"
}
```

Example Scenario

DELETE https://api.example.com/users/123

Making the API Call

- With the request set up, the client sends it to the API server using a tool or library that can handle HTTP requests.

- In Android development, using Kotlin, the HTTP methods GET, POST, PUT, and DELETE are crucial for interacting with web APIs. The popular library **Retrofit** is commonly used to streamline the process of making these API calls.

Retrofit Setup in Kotlin

First, you need to set up Retrofit in your project:

- Add Dependencies:
  - implementation 'com.squareup.retrofit2:retrofit:2.9.0'
  - implementation 'com.squareup.retrofit2:converter-gson:2.9.0'

Retrofit Setup in Kotlin – Define a Service Interface:

```
interface UserService {
    @GET("users")
    fun listUsers(): Call<List<User>>

    @GET("users/{id}")
    fun getUser(@Path("id") userId: Int): Call<User>

    @POST("users")
    fun createUser(@Body user: User): Call<User>

    @PUT("users/{id}")
    fun updateUser(@Path("id") userId: Int, @Body user: User): Call<User>

    @DELETE("users/{id}")
    fun deleteUser(@Path("id") userId: Int): Call<Void>
```

Retrofit Setup in Kotlin – Initialize Retrofit:

```
val retrofit = Retrofit.Builder()
    .baseUrl("https://api.example.com/")
    .addConverterFactory(GsonConverterFactory.create())
    .build()

val userService = retrofit.create(UserService::class.java)
```

Example Usage of HTTP Methods

```
//GET: Retrieve a list of users or a specific user.
val call = userService.listUsers()
call.enqueue(object : Callback<List<User>> {
    override fun onResponse(call: Call<List<User>>, response: Response<List<User>>) {
        if (response.isSuccessful) {
            // Use the list of users
        }
    }
    override fun onFailure(call: Call<List<User>>, t: Throwable) {
        // Handle failure
    }
})
```

Example Usage of HTTP Methods

```
//POST: Add a new user.val call = userService.listUsers()
val newUser = User(name = "Jane Doe", email = "jane.doe@example.com")
val call = userService.createUser(newUser)
call.enqueue(object : Callback<User> {
    override fun onResponse(call: Call<User>, response: Response<User>) {
        if (response.isSuccessful) {
            // User created
        }
    }
    override fun onFailure(call: Call<User>, t: Throwable) {
        // Handle failure
    }
})
```

Example Usage of HTTP Methods

```
//PUT: Update an existing user's details.
val updatedUser = User(name = "Jane Roe", email = "jane.roe@example.com")
val call = userService.updateUser(123, updatedUser)
call.enqueue(object : Callback<User> {
    override fun onResponse(call: Call<User>, response: Response<User>) {
        if (response.isSuccessful) {
            // User updated
        }
    }
    override fun onFailure(call: Call<User>, t: Throwable) {
        // Handle failure
    }
})
```

Example Usage of HTTP Methods

```
//DELETE: Remove a user.
val call = userService.deleteUser(123)
call.enqueue(object : Callback<Void> {
    override fun onResponse(call: Call<Void>, response: Response<Void>) {
        if (response.isSuccessful) {
            // User deleted
        }
    }
    override fun onFailure(call: Call<Void>, t: Throwable) {
        // Handle failure
    }
})
```

Let’s use Google Maps API as an Example

- This app will help users explore "hidden gems" around Seattle, WA, showcasing how to integrate and interact with Google Maps within an app.

Let’s use Google Maps API as an Example

Step 1: Set Up Your Project

- Create a New Android Project:
  - Open Android Studio and start a new project.
  - Choose ”Empty Views Activity" for simplicity.
- Add Google Maps Dependencies:
  - Open your build.gradle (Module: app) file.
  - Add the following line to include the Google Maps API dependency:
  - implementation 'com.google.android.gms:play-services-maps:17.0.1'

Let’s use Google Maps API as an Example

Step 2: Add Permissions and Generate API Key

- Add Internet Permission:
  - In your AndroidManifest.xml, add the following permission:
    - <uses-permission android:name="android.permission.INTERNET"/>
- Get a Google Maps API Key:
  - Visit the Google Cloud Console, create a new project, and enable the Google Maps API.
  - Generate an API key which will be used to authenticate your application with Google services.
  - Add the API key to your AndroidManifest.xml:
    - <meta-data
      android:name="com.google.android.geo.API_KEY”
      android:value="YOUR_API_KEY"/>

Let’s use Google Maps API as an Example

Step 3: Integrate Google Maps

- Modify the layout XML file (activity_main.xml):
  - Include a Fragment that will display the map:
  - <fragment
    android:id="@+id/map”
    android:name="com.google.android.gms.maps.SupportMapFragment”
    android:layout_width="match_parent”
    android:layout_height="match_parent"/>
  - This XML fragment embeds a map into the activity layout.

Let’s use Google Maps API as an Example

Step 3: Integrate Google Maps

- Initialize the map in your Activity (MainActivity.kt):
  - val mapFragment = supportFragmentManager
  - .findFragmentById(R.id.map) as SupportMapFragment
  - mapFragment.getMapAsync { googleMap ->
  - // This is where we use the Google Maps API
  - setUpMap(googleMap)
  - }

Let’s use Google Maps API as an Example

Step 4: Customize the Map for Seattle, WA

- Configure Map Settings (MainActivity.kt):
  - fun setUpMap(map: GoogleMap) {
  - map.apply {
  - mapType = GoogleMap.MAP_TYPE_NORMAL
  - uiSettings.isZoomControlsEnabled = true
  - moveCamera(CameraUpdateFactory.newLatLngZoom(LatLng(47.6062, -122.3321), 12f))
  - }
  - }

Let’s use Google Maps API as an Example

Step 5: Add Markers for Hidden Gems

- Add Markers on the Map (MainActivity.kt):
  - Define a list of coordinates representing hidden gems and add markers:
    - val hiddenGems = listOf(
    - LatLng(47.6101, -122.3421), // Sculpture Park
    - LatLng(47.6289, -122.3426) // Gas Works Park
    -)
    - hiddenGems.forEach { location ->
    - map.addMarker(MarkerOptions().position(location).title("Hidden Gem"))
    - }

Let’s use Google Maps API as an Example

Step 6: Interactivity and Testing

- Handle Marker Click Events:
  - Provide more details when a marker is tapped:
    - map.setOnMarkerClickListener { marker ->
    - // Code to display an info window or another activity with details
    - false
    - }