Android Development

Chapter 7 - REpresentational State Transfer

katholieke hogeschool
associatie KU Leuven
vives

Android Development
Chapter 7 - REpresentational State Transfer

REST

katholieke hogeschool
associatie KU Leuven
vives

REpresentational State Transfer (REST)

- REpresentational State Transfer (REST) is the software architectural style of the World Wide Web.
  - Coordinated set of constraints to the design of components in a distributed hypermedia system
  - Can lead to a higher-performing and more maintainable architecture.

- RESTful systems are systems that conform to the constraints of REST
- RESTful systems typically, but not always, communicate over HTTP with the same HTTP verbs
  - GET, POST, PUT, DELETE, etc.
  - Used by web browsers to retrieve web pages and to send data to remote servers.
- REST interfaces with external systems using resources identified by URI
  - for example /people/tom
  - can be operated upon using standard verbs, such as DELETE

Constraints

- **Client–server**
  - A uniform interface separates clients from servers
  - Clients are not concerned with data storage, which remains internal to each server
  - Servers are not concerned with the user interface or user state

- **Stateless**
  - No client context is being stored on the server between requests
  - Each request from any client contains all the information necessary to service the request, and session state is held in the client
  - The session state can be transferred by the server to another service such as a database to maintain a persistent state for a period and allow authentication

katholieke hogeschool vives
associatie KU Leuven

Constraints

- **Client–server**
  - A uniform interface separates clients from servers
  - Clients are not concerned with data storage, which remains internal to each server
  - Servers are not concerned with the user interface or user state

- **Cacheable**
  - As on the World Wide Web, clients and intermediaries can cache responses
  - Responses must therefore, implicitly or explicitly, define themselves as cacheable, or not, to prevent clients from reusing stale or inappropriate data in response to further requests
  - Well-managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance

Constraints

- **Stateless**
  - Essentially, what this means is that the necessary state to handle the request is contained within the request itself, whether as part of the URI, query-string parameters, body, or headers
  - The URI uniquely identifies the resource and the body contains the state (or state change) of that resource
  - Then after the server does it's processing, the appropriate state are communicated back to the client via headers, status and response body
  - The client must include all information for the server to fulfill the request, resending state as necessary if that state must span multiple requests.
  - Statelessness enables greater scalability since the server does not have to maintain, update or communicate that session state.
    - Additionally, load balancers don't have to worry about session affinity for stateless systems.

Constraints

- **Layered system**
  - A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way
  - Intermediary servers may improve system scalability by enabling load balancing and by providing shared caches.
  - They may also enforce security policies.

- **Code on demand (optional)**
  - Servers can temporarily extend or customize the functionality of a client by the transfer of executable code.
    - Examples of this may include compiled components such as Java applets and client-side scripts such as JavaScript.
  - "Code on demand" is the only optional constraint of the REST architecture.

katholieke hogeschool vives
associatie KU Leuven

Constraints

- **Uniform interface**
  - The uniform interface constraint is fundamental to the design of any REST service.
  - The uniform interface simplifies and decouples the architecture, which enables each part to evolve independently.
  - The four constraints for this uniform interface are:
    - **Identification of resources**
      - Individual resources are identified in requests, for example using URIs in web-based REST systems.
      - The resources themselves are conceptually separate from the representations that are returned to the client.
      - For example, the server may send data from its database as HTML, XML or JSON, none of which are the server's internal representation.

Constraints

- **Manipulation of resources through these representations**
  - When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource.
    - » Provided it has the permissions to do so

- **Self-descriptive messages**
  - Each message includes enough information to describe how to process the message.
  - For example, which parser to invoke may be specified by an Internet media type (previously known as a MIME type).
  - Responses also explicitly indicate their cacheability

Constraints

- **Hypermedia as the engine of application state (HATEOAS)**
  - Clients deliver state via body contents, query-string parameters, request headers and the requested URI (the resource name).
  - Services deliver state to clients via body content, response codes, and response headers.
  - This is technically referred-to as hypermedia (or hyperlinks within hypertext).

  - Aside from the description above, HATEOAS also means that, where necessary, links are contained in the returned body (or headers) to supply the URI for retrieval of the object itself or related objects.

katholieke hogeschool vives
associatie KU Leuven

Android Development
Chapter 7 - REpresentational State Transfer

Making REST requests from Android

katholieke hogeschool vives
associatie KU Leuven

Example
Github

- You can make a simple GET request to for example github
- Try it out
  - https://api.github.com/repos/square/retrofit/contributors
  - Just open browser and surf to this URI

Retrofit

- A type-safe HTTP client for Android and Java
- Makes it incredibly easy to make REST request to RESTful server app

- Retrofit turns your HTTP API into a Java interface.

```
public interface GitHubService {
    @GET("repos/{owner}/{repo}/contributors")
    Call<List<Contributor>> contributors(
            @Path("owner") String owner,
            @Path("repo") String repo
    );
}
```

  – No slash at the start of the resource uri

Retrofit

- The Retrofit class generates an implementation of the GitHubService interface.
  - You can do this for example in your onCreate() method when your activity may send multiple REST requests to different URI's

```
Retrofit retrofit = new Retrofit.Builder()
        .baseUrl("https://api.github.com/")
        .build();

GitHubService service = retrofit.create(GitHubService.class);
```

- Slash at the end of the base url

Retrofit

- Create a representation of the contributor
  - This is placed in its own separate class file !

```
public class Contributor {
    // @Expose = An annotation that indicates this member
    // should be exposed for JSON serialization or deserialization.
    @Expose
    private String login;

    @Expose
    private int contributions;

    public String getLogin() {
        return login;
    }

    public int getContributions() {
        return contributions;
    }
}
```

Retrofit

- Making the request and getting the result

```
// Params needed for request
String owner = "square";
String repo = "retrofit";

// Create call instance
Call<List<Contributor>> call = service.contributors(owner, repo);

// Call enqueue to make an asynchronous request
call.enqueue(new Callback<List<Contributor>>() {
    // On Android, callbacks will be executed on the main thread
    @Override
    public void onResponse(Response<List<Contributor>> response, Retrofit retrofit) {
        if (response.body() != null) {
            List<Contributor> contributors = response.body();
        } else {
            Log.e("REST", "Request returned no data");
        }
    }

    @Override
    public void onFailure(Throwable t) {
        Log.i("REST", t.toString());
        Toast.makeText(getApplicationContext(), "Request failed", Toast.LENGTH_SHORT).show();
    }
});
```

Retrofit

- To be able to use retrofit you need to add a couple of dependencies to your gradle script (Module app)

```
dependencies {
    //...
    compile 'com.squareup.retrofit:converter-gson:2.0.0-beta2'
    compile 'com.squareup.retrofit:retrofit:2.0.0-beta2'
}
```

  – Make sure to rebuild your project after this

katholieke hogeschool vives
associatie KU Leuven

Github RESTful API

- Try to make the previous code work with the Github service
- Try to save an additional attribute inside your class
- Display the information on the GUI after a successful request

Android Development
Chapter 7 - REpresentational State Transfer

Now for the Thumper Control

katholieke hogeschool vives
associatie KU Leuven

Thumper RESTful Control API

NeoPixel Control

- To get started I created a simple node.js application (ServerSide JavaScript) to control NeoPixels on the Thumper
- Clone the application from GitHub
  - https://github.com/BioBoost/node_thumper_control.git

- Install node.js on your computer
- Go to the directory and execute "npm install"
- Run the app using "node thumper_app.js"

katholieke hogeschool associatie KU Leuven vives

Thumper RESTful Control API

NeoPixel Control

- Some resources are:

```
// @GET neopixels/strings/:id
// returns { "string_id": "1", "number_of_pixels": "8" }

// @POST neopixels/strings/:id
// expects { "red": 10, "green": 255, "blue": 0 }
// returns { "status": "success" }
```

- You can test this using Chrome and a plugin called "Advanced Rest Client"

Thumper RESTful Control API

NeoPixel Control

- Start by creating a class called NeoPixelString with an "id" attribute and a "numberOfPixels"
- Next create a GUI in a separate activity to display the information and to initiate the REST request
- Define the NeoPixel service interface with the GET route
  - Add the POST later

```
// @GET neopixels/strings/:id
// returns { "string_id": "1", "number_of_pixels": "8" }
```

- Setup retrofit and an instance of the NeoPixel service
  - Preferable in the onCreate() method of the activity

- Make the request and display the info

Thumper RESTful Control API

NeoPixel Control

- Create a NeoPixelColor class with three attributes (red, green and blue)
  - Add both a default and an initialization constructor

- Add the POST route to the NeoPixel interface

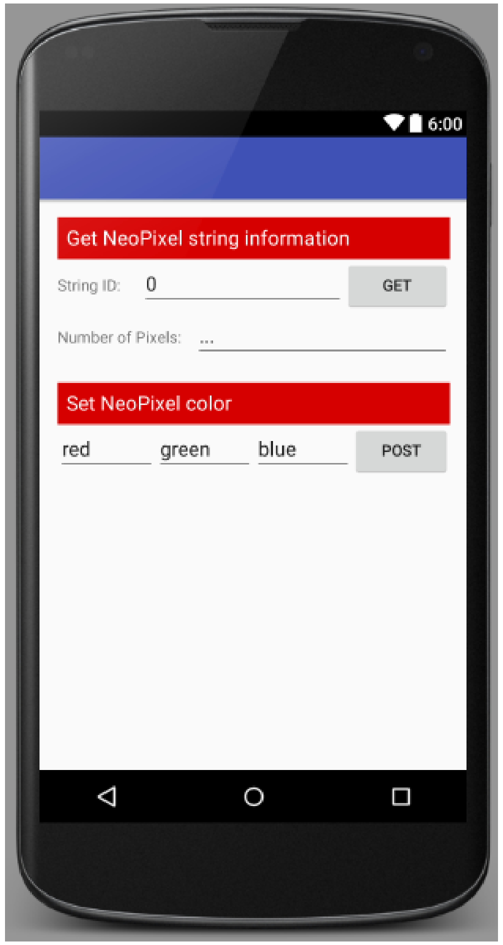
```
// @GET neopixels/strings/:id
// returns { "string_id": "1", "number_of_pixels": "8" }
```

- Before making the request, create an instance of the NeoPixelColor class and supply it to the request
  - Start with a hardcoded color
  - Next create a GUI to let the user change the color
    - Add validations to check the values are in range [0, 255]

Thumper RESTful Control API

NeoPixel Control

- My example of the GUI
  - Make sure to create your own and be creative


6:00
Get NeoPixel string information
String ID: 0
GET
Number of Pixels: ...
Set NeoPixel color
red
green
blue
POST