Instant mobile receiver

Building a service on top of the Amazon Cloud Services

Lukáš Pastuszek

Bachelor's thesis
December 2016
School of Technology, Communication and Transport
Bachelor's Degree Program in Information and Communications Technology

jamk.fi

Description

Author(s) Pastuszek Lukáš	Type of publication Bachelor's thesis	Date December 2016
	Number of pages 45	Language of publication: English
		Permission for web publication: yes
Title of publication **Instant mobile receiver** Building a service on top of the Amazon Cloud Services		
Degree programme Information and Communications Technology		
Supervisor(s) Salmikangas Esa		
Assigned by		

Description

This paper describes the design, implementation and deployment of a 3-application system including the information of used technologies, architectures and services.

The frontend application is a static web application interface written in HTML, JavaScript, CSS3, and it is used to send content to a mobile device from computer/laptop. The design of the interface is very easy, straight-forward and user-friendly. It uses HTML5 Drag&Drop functionality for delivering files and text area for delivering text. It is based on Angular.js framework, using Grunt for task automatization and Bower for handling dependencies on other libraries. It runs as a static web page deployed to Amazon S3 bucket.

The web service application was developed on top of Node.js JavaScript server using the Restify framework. It is a multilayered web service, which provides resources to the client application using REST API interface. It was developed and deployed to Amazon Web Services (AWS) cloud inside of Docker container. It is designed to use the DynamoDB NoSQL database as a part of Amazon cloud services.

The system also uses another AWS service S3 for file storage, AWS API Gateway as a proxy authentication layer for the file storage and AWS Lambda as an API Authorizer.

The mobile application is targeted for Android operation systems supporting Android SDK 15 and higher. The mobile phone with the application installed is able to receive and display content sent by the user through the web interface.

Keywords (subjects)
AWS, Cloud development, REST, Node.js, Angular, mobile application, Android, Lambda function, DynamoDB

Miscellaneous

Table of contents

List of figures

List of abbreviations

AWS	Amazon Web Services
AWS CLI	Amazon Web Service Command Line Interface
CORS	Cros-Origin Resource Sharing
CSS	Cascading Style Sheets
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
IaaS	Infrastructure as a Service
IDE	Integrated Development Environment
JS	JavaScript
MVC	Model-View-Controller
OS	Operating System
PaaS	Platform as a Service
SDK	Software Development Kit
URL	Uniform Resource Locator

1. Introduction

1.1. Motivation

The purpose of creating the application was to offer an alternative to delivering content to the user’s own mobile device (smartphone, tablet) instantly. There are several ways how to insert content into a person’s own mobile device:

- using wired connection to the computer – usually USB interface,
- using the direct wireless connection to the computer - usually Bluetooth,
- using third party cloud services – iCloud, Dropbox, OneDrive or others.

Using the first two options does not seem very modern these days because it requires setting up a direct connection between a computer and a mobile. Cloud services seem like an easier and faster option. These cloud solutions usually offer more interfaces to insert and download content from their servers (web based, desktop application, mobile application). The process of sending content to the mobile device consists of an uploading file into the cloud, waiting for synchronization, accessing the cloud from the mobile device and downloading the file.

This project offers another alternative how to deliver the content faster and more easily. The content is delivered to the mobile application via the internet by notification so the information is available on the phone immediately.

1.2. High-level design

The project is divided into three separate parts, which are connected together. The foundation of the project is a web service providing REST API for sharing resources and data for both web and mobile applications which act here as two different clients with different functionalities. The concept is designed in the way that all the applications are completely separate, which allows complete substitution of one or more applications in the future by new a one. This would, for instance, allow adding another client for additional mobile platform fairly easily. The high-level diagram of the whole system can be seen in Figure 1.

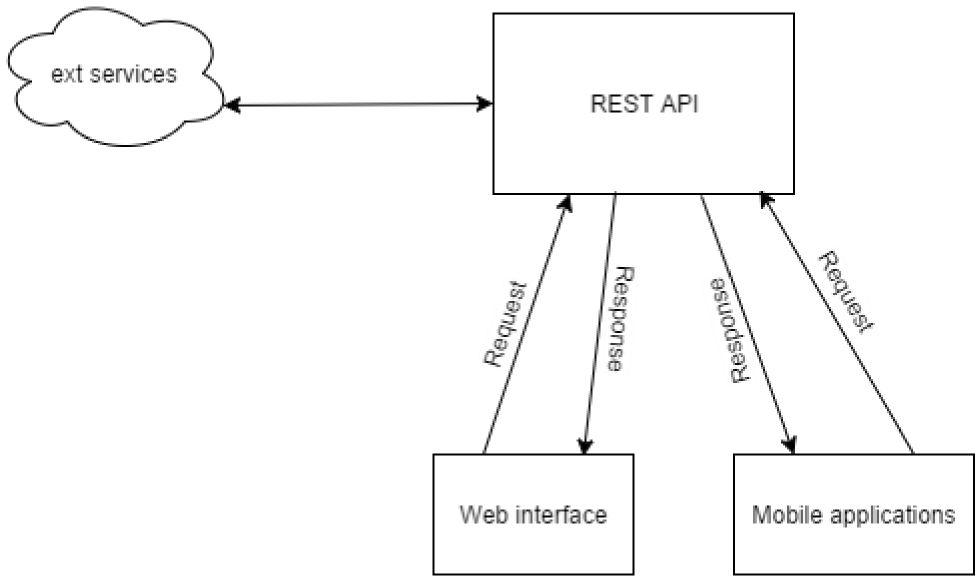


Figure 1 - High-level design of the system

1.3. Thesis structure

Chapter 1 introduces the motivation and, basic description, requirements and design of the complete service. Each of the following three chapters are dedicated to a single application of the system, therefore, the description of building the web service is addressed in chapter 2, creating the web interface in chapter 3 and developing the mobile application in chapter 4. Chapter 5 describes the implementation of sending text notifications from the web client and chapter 6 sending notifications content from a text file including the description how to use third-party AWS S3 as a file storage and the API Gateway as an authentication proxy service.

1.4. Requirements

1.4.1. Functional requirements

This projects requirement is to create a prototype of the service. In the first phase which this paper describes the application will have the following functional requirements:

- a mobile device will be able to sign up for receiving notifications,

- a user will be able to see his devices and will be able to choose to which of those the content will be sent,
- a user will be able to send a text to the device from the web interface
- a user will be able to send a text file content to the device from the web interface
- a mobile device will be able to display notifications from an incoming data
- all parts of the application need to have a login option, no guest users are being allowed

1.4.2. Non-functional requirements

A user needs to be able to access only the content he is permitted to see and a user needs to be able to send content only to his own devices. This must be enforced in the back-end service.

All parts of the services must send all requests with the correct authorization protocol. Consuming resources of the web service with unauthorized users should not be possible.

Service availability should be high enough so the service can be evaluated as reliable, easy scalability of the service must be easy to perform and the capacity of the computational power needs to be high enough for the future usage.

2. Web service backend application

2.1. REST

“Web services are purpose-built web servers that support the needs of a site or any other application” (*REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*, 2016). Application Programming Interface (API) is a part of the web service, which exposes data and functions to communicate with other systems. When REST is discussed Representational State Transfer is meant. That is a name Roy Fielding gave to this architectural API style in his dissertation thesis and which is now one of the most common practices how to build web services.

The most important aspects of the REST architecture are

– separation of a client and a server,
– stateless communication,
– layered system architecture.

Communication must be stateless in the way that the requests to the server container all the information needed to process the requests and respond accordingly. Layered system architecture means that the layers communicate only with the adjacent layers, but do not know the complete architecture of the system. (*Architectural Styles and the Design of Network-based Software Architectures, 2016)*

2.2. Architecture

The web service described throughout this chapter runs in the cloud on top of a JavaScript server platform Node.js using framework Restify to expose REST API. It has its own architecture because it works like a standalone application independent of each other. It is divided into layers, which only communicate with the layers adjacent to it. Those layers are:

- API layer,
- controllers,
- model.

API layer or also Routes is the first layer which the incoming requests land to. It resolves the path in the URL and sends the request data to the next layer. Controllers are components of the second layer which include most of the logic. The first layer decides which controller should process the request, and the controller itself handles it. It consumes other layers which are bonded to it e.g. the model layer (for database access) or different components in forms of libraries or other dependent third party services. Model layer is the layer to provide access to the database, it is an interface between the database driver and the controller.

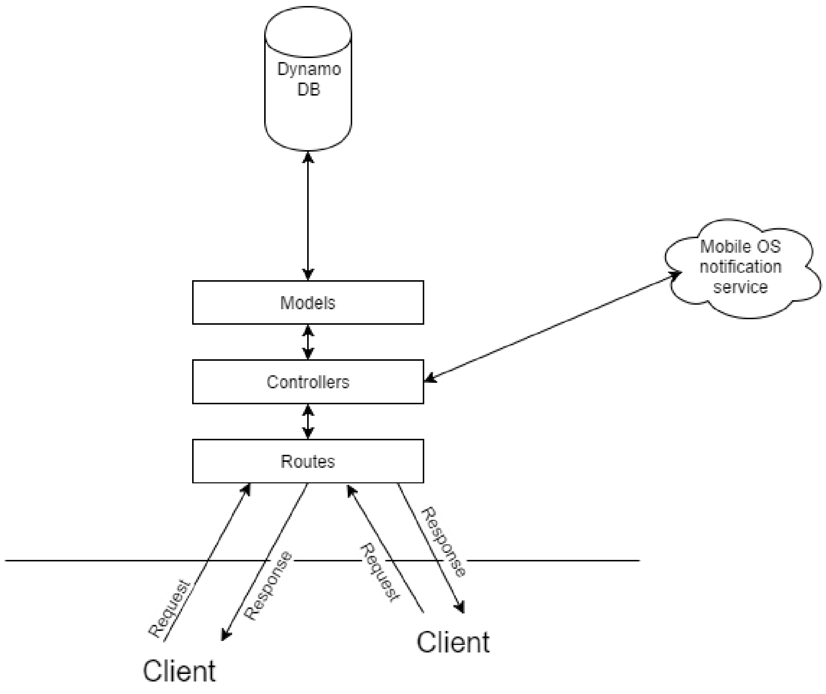


Figure 2 Architecture of the Web Service

2.3. Technologies

2.3.1. Web server

There are of course very many options what web server to use, and this important decision needs to be resolved before the implementation begins. The service is written in the Node.js programming language because of its high and still increasing popularity, large amount of 3rd party libraries, performance reason and its ability to realize non-blocking asynchronous operations to external services.

Restify is a JavaScript framework used to develop the service. It is built on top of Node.js platform. It is a more lightweight alternative compared to the other even more popular framework called Express.js. It does not support some features which Express.js does, e.g. rendering views and templating functionality. This makes Restify not ideal for MVC applications but more suitable for building Web services with REST API.

2.3.2. Persistent storage

Persistent storage, usually a database is a very important part of almost every application. A trend is to use NoSQL databases over more classical SQL databases like Oracle, Microsoft SQL Server or MySQL.

In the world of NoSQL databases, there are many platforms to choose from – some of the most popular being MongoDB or CouchDB. The services are connected to DynamoDB because it is available in the same Amazon cloud where the service will be running as well.

DynamoDB is a NoSQL database, which uses documents data model. It supports storing, querying and updating documents stored in tables. The AWS DynamoDB service is not divided into databases. The tables within the cloud service are separated by regions only. AWS currently has twelve regions of which four are located in the USA, two in Europe, five in Asia and one in South America. When accessing the table, the correct region needs to be selected while making the request/connection to the database. A similar logic applies to most of the Amazon cloud services.

Definition of the database table is given by the following:

- table name,
- key schema,
- attribute definitions,
- secondary indexes,
- throughput.

The key schema specifies a primary key for the given table. It must include hash attribute and optionally also range attributes. It is possible to query and get data from the table by these values. Attribute definition is a specification of the attributes used in the key schema and in secondary indexes. Possible types are only string, number or binary. Secondary indexes are additional indexes which allow querying the database using different attributes than only the key schema. Throughput specifies reading and writing reserved capacities for the table. Table with read and write capacity equal to one is able to read 4KB document per second and to write one document per second of size up to 1 KB. (Provisioned throughput, 2016)

Table structure

NoSQL databases usually do not have a strict data structure like SQL databases. The structure of the data model used by the web service is described in the following JSON:

```
User: {
        email: String,
        security: {
                passwordHash: String,
                salt: String
        }
}

Token: {
        email: String,
        // Only required when logging from mobile device
        mobileDevice: {
                deviceId: String,
                deviceName: String,
                firebaseToken: String
        },
        tokenHash: String
}

Content: {
        id: String,
        email: String,
        deviceId: String,
        firebaseToken: String,
        text: String,
        fileName: String,
        fileUrl: String,
        type: String
}
```

Primary key

DynamoDB does not offer any possibility to autogenerate a primary key for the table. The web service needs to generate a unique id to store content using its own algorithm. The id for content is used throughout the web service and also sent to Android device to uniquely identify the content.

A good way to generate unique id is use a current timestamp. The string generated by the service is composed in the following manner:

```
function generateUniqueId(email) {
        let id = Date.now() + '_' + email + '_' + Math.floor(Math.random()
* 99999);
        return id;
}
```

The string is a composition of the Date.now() function which returns the milliseconds elapsed since 1 January 1970 00:00:00 UTC. Adding the email of the user would allow every user to send one request every one millisecond and the system would be still working properly. Adding a random number between 0 and 99 999 ensures that even more requests are very likely to be handled successfully even though this amount of requests per user is unreal.

2.4. Development environment

2.4.1. Sending requests to the service

To test the web service interface, a developer needs to send requests to the server and see the responses the service returns to the client. This requires some tool which is able to do that. The basic command line Unix utility is Curl. It supports many different protocols to send requests including all methods for HTTP and HTTPS.

Another client which was used for sending requests to the server while developing was Chrome application called Postman. It offers a graphical interface to design, save and manage the requests. It allows to select a method for the request, add header information to the request or attach body parameters.

2.4.2. Docker

One of the key technologies while developing the backend web service was using the Docker containers. The definition from official Docker website states that "Docker containers wrap a piece of software in a complete file system that contains everything needed to run: code, runtime, system tools, system libraries – anything that can be installed on a server. This guarantees that the software will always run the same, regardless of its environment." (What is Docker, 2016)

The Docker can be considered a lighter version of virtualization. It runs on top of the operation system but does not run another complete OS as normal virtualization technologies do. Instead of running an additional OS, it runs the Docker engine on top of hosting OS and the Docker container on top of that.

The reason why Docker is getting more and more popularity is its environment independence. To be able to run the code the only thing needed is to have the Docker itself installed. It brings the advantage of having installation instructions for all the dependencies in the container as a part of the code-base. This also applies to deploying the container to the cloud or to the server. All that is needed is to make sure that the target platform supports deploying a Docker container.

Docker container

Docker container is a running instance of a Docker image. One of the important features of Docker is the Docker Hub which is a repository for Docker images where users can find many free available images to use or build their own images on top of those.

To run a Docker container a Dockerfile with the image definition is needed. Docker builds the image according to the definition in the file. Dockerfile for the service is presented in the following code block.

```
FROM node

# update packages for package manager
RUN apt-get update


ENV NVM_DIR /usr/local/nvm
ENV NODE_VERSION 4.3.1

RUN npm config set registry http://registry.npmjs.org/

# Install nvm with node and npm
RUN curl
https://raw.githubusercontent.com/creationix/nvm/v0.30.1/install.sh | bash
\
    && . $NVM_DIR/nvm.sh \
    && nvm install $NODE_VERSION \
    && nvm alias default $NODE_VERSION \
    && nvm use default

ENV NODE_PATH $NVM_DIR/v$NODE_VERSION/lib/node_modules
ENV PATH      $NVM_DIR/versions/node/v$NODE_VERSION/bin:$PATH
```

```
RUN npm install -g node-inspector

# Create app directory
RUN mkdir -p /usr/src/app

WORKDIR /usr/src/app

COPY package.json /usr/src/app/
RUN npm install

COPY . /usr/src/app

EXPOSE 8008 8000 8080 5858 443


CMD [ "node", "src/server.js" ]
```

The Dockerfile showed above inherited the public image called “node” from the Docker Hub. The image is extended by the Unix commands to use NVM utility to manage custom Node.js versions, install NPM global and local dependencies according to the package.json file and expose ports from the container so the web server running in the container is available from the operating system.

Docker Compose

Docker Compose is a tool for defining and running Docker containers. It allows to set up running configuration for the container in the Docker-compose.yml as shown in the following code block.

```
version: '2.0'

services:
  rest-api:
    build: .
    command: node src/server.js
    image: instant-mobile-receiver-api
    ports:
      - "8008:8008"
      - "8080:8080"
      - "5858:5858"
      - "443:443"
    volumes:
     - .:/usr/src/app
    env_file: .env
    links:
     - local-db
```

All the options for running the container are defined in the file. The file can include many container definitions which can be linked to each other. The file defines the named container called “rest-api” which can be run with the following command.

```
docker-compose up rest-api
```

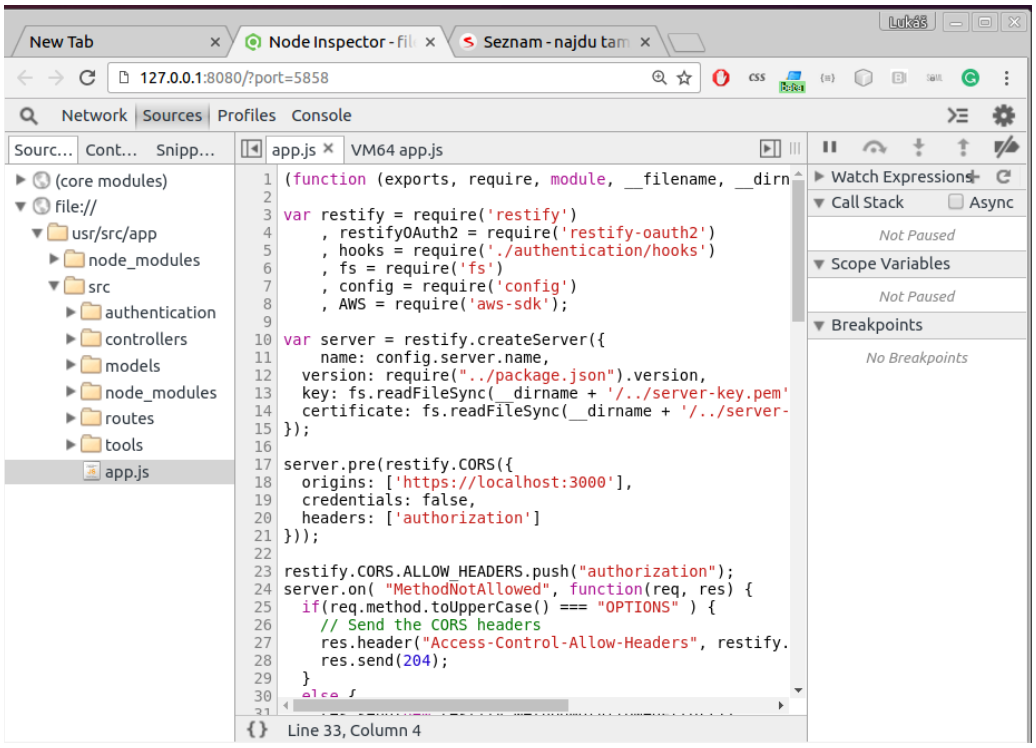
2.4.3. Debugging

Debugging of the Node.js applications is not that comfortable, the default debugger implementation is a command line interface which is not very easy to use because of plain text commands and no graphic interface.

When doing a front-end JavaScript application, the code is run in the browser. Browsers usually offer they own debugging and developing tools. The code for this service was written in a software called Sublime Text 3 which is a very sophisticated text editor suitable for very many tasks including coding. It offers many free installable plug-ins which simplify the development, however, it still does not make it full IDE. Debugging inside Sublime Text is therefore not possible.

Node inspector

The solution for debugging was to use Node inspector. It is an interface which allows debugging Node.js applications in the web browser. It uses Blink Developer Tools which is a visually very similar interface to the developer tools in Chrome web browser. The user interface can be seen in Figure 3.



```
(function (exports, require, module, __filename, __dirn

var restify = require('restify')
    , restifyOAuth2 = require('restify-oauth2')
    , hooks = require('./authentication/hooks')
    , fs = require('fs')
    , config = require('config')
    , AWS = require('aws-sdk');

var server = restify.createServer({
    name: config.server.name,
  version: require("../package.json").version,
  key: fs.readFileSync(__dirname + '/../server-key.pem'
  certificate: fs.readFileSync(__dirname + '/../server-
});

server.pre(restify.CORS({
  origins: ['https://localhost:3000'],
  credentials: false,
  headers: ['authorization']
}));

restify.CORS.ALLOW_HEADERS.push("authorization");
server.on( "MethodNotAllowed", function(req, res) {
  if(req.method.toUpperCase() === "OPTIONS" ) {
    // Send the CORS headers
    res.header("Access-Control-Allow-Headers", restify.
    res.send(204);
  }
  else {
```

Figure 3 Node inspector interface

The Node Inspector runs as a website on the server when the application is run with command `node-debug <filename.js>` instead of `node <filename.js>` which is used to run node JS file. The default URL to access the Node inspector is 127.0.0.1:8080/?port=5858.

2.5. Running environment

2.5.1. Possible environments

When designing the environment, the following options exist:

- Own server solution,
- Infrastructure as a Service (IaaS),
- Platform as a Service (PaaS),

These three options are different in terms of costs, computational power and effort to make the running environment set up.

A server solution of its own brings the most of the freedom; however, it is the most difficult to set up. It all starts with buying the hardware and includes setting up the operating system, installing all needed dependencies, setting up networking infrastructure and deploying the application in the end. This is a time and cost consuming option not suitable for a relatively small web service.

Infrastructure as a Service provides the infrastructure for the application. It is usually in the form of a dedicated or virtualized server to which user has remote access. Still, while setting up this environment all the dependencies, libraries needs to be set up while deploying.

Platform as a Service provides the user with the environment the way that deployment can be done either automatically or with a small amount of effort. It can be as simple as only uploading the code to the server, committing the code to the repository; however, of course it can be slightly more complex.

2.5.2. Deployment to cloud

The web service is deployed to Amazon Web Services (AWS) platform which offers a variety of services used by the application.

Web server

As this web service application uses Docker container it needs a suitable service which would support running Docker container. The service is called E2 and it is a web server allowing to deploy code in many programming languages like Java, .NET, Python, JavaScript. It also supports deploying the Docker container.

Docker is a technology which lets a user set up the container before deploying and then installs all the dependencies inside the Docker container as specified in the configuration Dockerfile. EC2 is normally a classic example of PaaS because it provides a user with the finalized environment for deploying the code. When deploying a Docker image into EC2, the user is deploying slightly more than just a code. The user deploys the whole Docker settings, and therefore, it can also be considered an IaaS.

Manage environments

As the AWS offer many services, it is sometimes difficult, especially in the beginning to figure out which services to use and how to bind them together. That is why the Elastic Beanstalk is one of the services Amazon provides. It is a tool for deploying and managing different applications and application's environments without studying the service infrastructure too much in depth. It can automatically solve and integrate problems like provisioning, load balancing, scaling and health monitoring at one place. (What is AWS Elastic Beanstalk, 2016)

Automatic deployment

As Elastic Beanstalk is very good for managing and deploying, it does not possess a possibility to make a deployment from GitHub repository which is a platform where the code for the application is stored. AWS offers, therefore, another tool called CodePipeline which offers a possibility to pair the AWS with GitHub account and access public or private repositories directly. CodePipeline creates a pipeline which is paired to the branch of the selected repository. The deployment is run when a developer commits and pushes or merges changes into the branch specified in the pipeline. The pipeline is in this case configured to start automatic deployment into the Elastic Beanstalk service which is actually making the deployment into EC2 server.

2.6. Web service user authentication

2.6.1. Registration

To register a new user a request needs to be sent to path api/user/register by POST method. The body of the request needs to include two parameters – email and password. The web service validates if the user does not exist already and sends a success or error response back to the request initiator. This is the only type of request which does not need any special authentication.

2.6.2. Log in

When the user with the email and password is registered it is possible to log in. The path for the POST request is api/user/login. The body has to have three parameters – username, password and grant_type when authenticating a client from the web client. The grant_type needs to be specified as value “password” which tells the web service the user is logging with his username/email and password combination. Some more (e.g. 3rd party) log in options can and probably will be implemented in the future for easier registration and logging in.

While logging in to the server from a mobile client the request needs additional parameters specifying the device id, device name and token included in the body part of the request.

The same request must also include authorization header. Every platform has its own username and password to use with HTTP Basic access authentication. These two values in form username:password are encoded with the Base64 algorithm and inserted as Authentication header value as a string with prefix “Basic “. Thanks to that, the web service can recognize what client the user is logging in from, which can be useful in some cases. The response after successful signing in request returns the Bearer token.

2.6.3. Authenticated request

When the client gets his Bearer token it needs to include the token with prefix “Bearer “ as an authorization header while performing any authenticated request to the server including the logout.

During authentication process in the web server the token is validated and if the validation is successful it resolves the tokens into email/username of the user who sends the request, which means that every request can be easily assigned to the user.

For resolving the token into email/username the server uses the caching-first approach. If the token is not found in the cache it tries to find it in the database. If resolving from neither cache nor database was not successful, the request responses with HTTP 401 Unauthorized header.

3. Web application

3.1. Design

While designing the graphical interface of the web application the focus was placed on simplicity, user-friendliness, and so the result is especially easy to use. A very important aspect was to implement the HTML5 Drag & Drop feature for file upload. The page where to the content is transferred can be seen in Figure 4. The page consists of the following elements:

- list of available devices,
- Drag&Drop space for files,
- text input for sending content,
- logout button,
- send button.

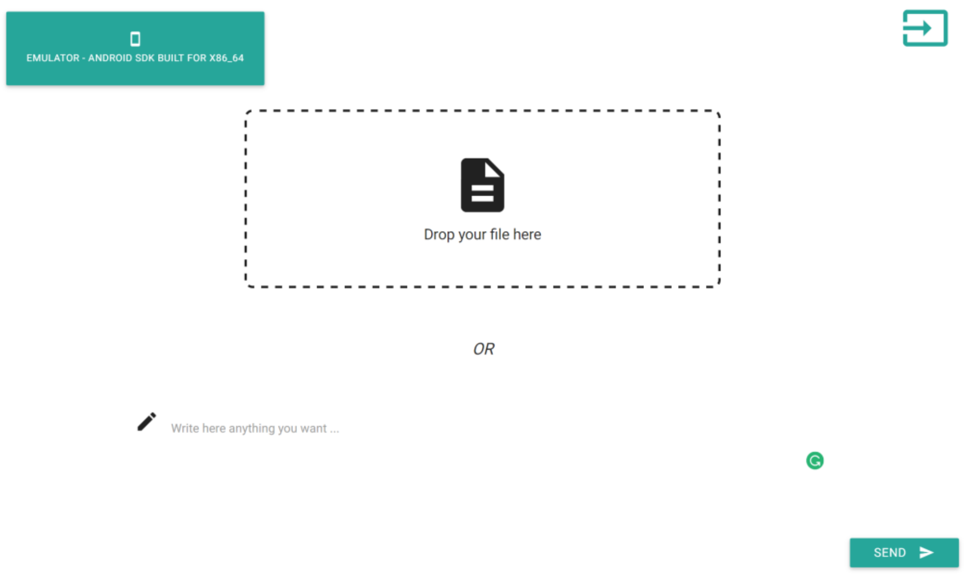

EMULATOR - ANDROID SDK BUILT FOR X86_64
Drop your file here
OR
Write here anything you want ...
SEND

Figure 4 Screenshot of the web interface

3.2. Architecture

The front end web interface is a JavaScript, HTML5, CSS application build on top of the Angular.js framework using various NPM packages. In the world of JavaScript front end technologies, new technologies are created almost daily and the trend what framework/library to use is changing exceptionally fast.

The application has a component-based architecture. The component is a standalone part of the application with dependencies. It is composed of two files – an HTML file and a JavaScript file. JS file contains the controller function, provides communication with other dependencies of the controller and specifies the data binding for the view. A component example is shown in Figure 5.

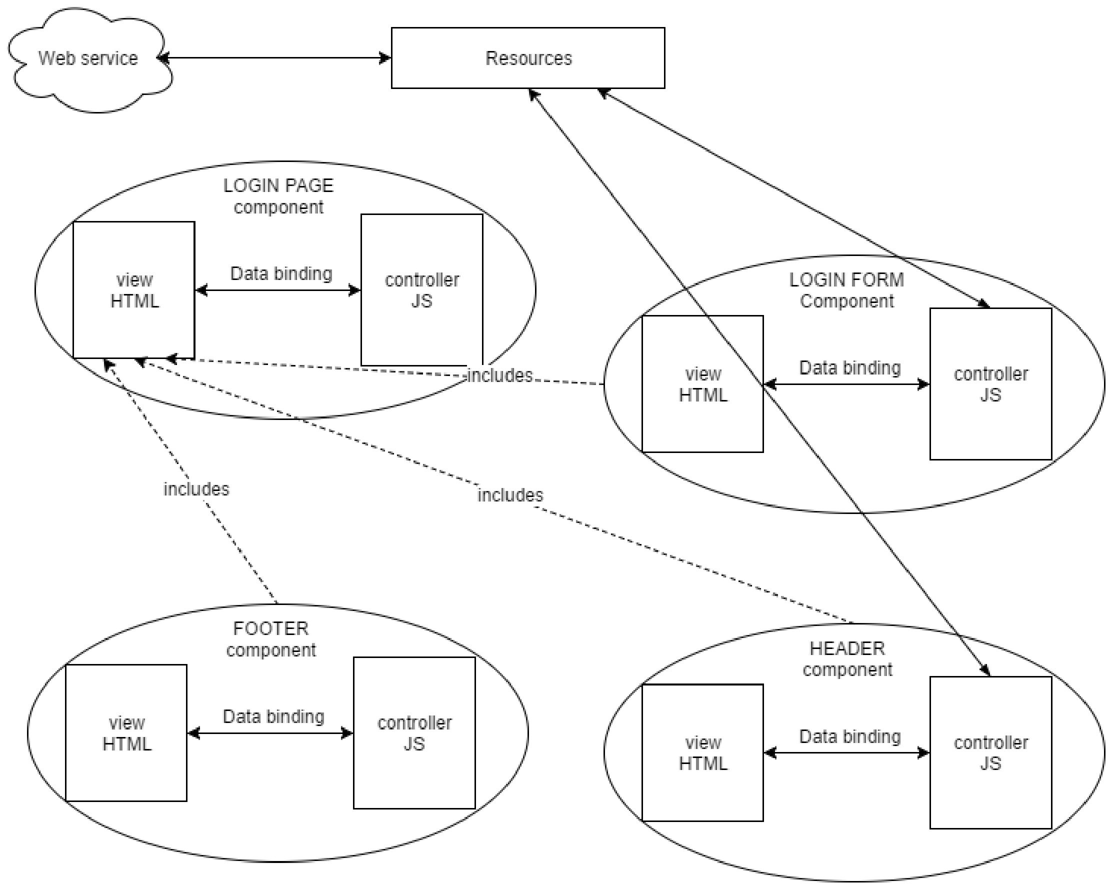


Figure 5 Front end application architecture

The controller's dependencies are third party libraries or services defined within the application. The services within the application form the so-called resource layer. The resource layer is the layer which is performing the communication with the web service. It sends requests to and receives responses from the web service. It provides the result for the component which handles it. Components designed in this way can be easily included together or can be reused in different parts of the application. The example of part of the application is visualized

3.3. Environment

3.3.1. Development environment

As part of a development environment, the application uses Grunt for performing automatization of tasks. Every task uses some Gulp Module which is available online for free using Bower or NPM package managers. The application uses Grunt for the following tasks:

- compiling the Less stylesheets into CSS3,
- create a static web server run on localhost.

3.3.2. Running environment

The outcome of this application is a completely static application without any server-side code. It is, therefore, possible to use much more light-weight production environment which can only hold static content without any actual computational power. The static content is hosted on Amazon S3 which is a storage service where any kind of data can be saved and retrieved. The data inside the service are organized into separate buckets, where every bucket has its name and settings. The settings include bucket permissions, logging, versioning and others.

One of the possibilities of the service is to use S3 as a static website hosting. It is aimed to host a static website which does not require any server-side processing or rendering.

Deployment

There are two possibilities to access the S3 bucket data. The first and most basic one is the web interface, where all the data and configuration options can be reviewed and changed. Deployment is possible by manually emptying the bucket and copying the files into it through the web interface.

The second possibility how to deploy the application is using the AWS CLI which is a complete set of tools to manage AWS services available for all major operation systems Windows, Linux and MacOS. To work remotely with the services, only two values AWSAccessKeyId and AWSSecretKey need to be specified as security credentials, e.g. set as environment variables of the shell. Deploying the code which in this context means simply uploading the files can be done with one command as stated in the following code block.

```
aws s3 sync <source-folder> s3://<bucket-name>
```

.

4. Mobile application

4.1. Android development

The main tool for developing an Android application is to use Android Studio IDE. The main programming language is Java using the Android SDK. An XML syntax is used to manage resources like layouts, styles and translations.

The main component of an Android application is an activity. It is a single screen the user can interact with. Each activity is independent even though they can pass the data between each other with intents. An activity is defined by its XML layout and its Java class.

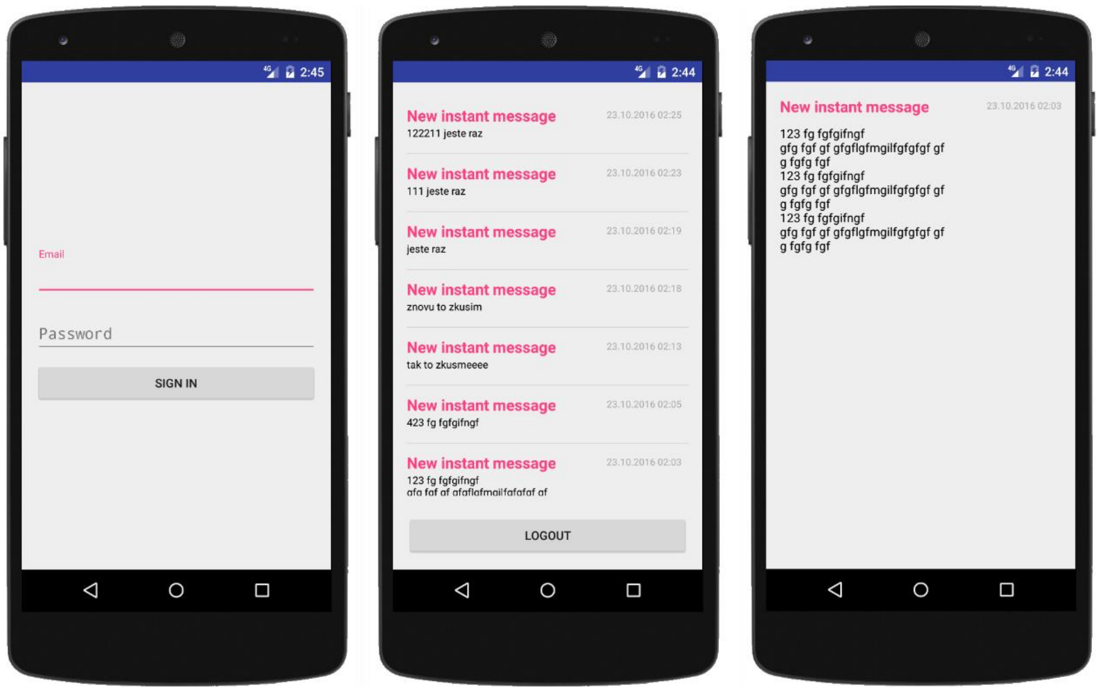
To test an Android application while developing, a part of the development kit is an emulator for Android devices. It is possible to download various versions of the Android operating systems and create virtual devices with different settings like OS version, device type, model name, screen resolution, memory and storage settings. The developers can run, test and debug their application with the emulator to make the development easier.

4.2. Application structure

The application has three activities listed here:

- login activity,
- homepage activity,
- detail activity.

Login activity is a screen where the user enters his email and password to register the device with his account, homepage activity provides the list of notifications which the device already received and offers a logout option for the user to register the device to another account. Detail activity displays the full content of the received data in case the content is too long. The screenshots of the views are shown in Figure 6.



New instant message 23.10.2016 02:25
122211 jeste raz

New instant message 23.10.2016 02:23
111 jeste raz

New instant message 23.10.2016 02:19
jeste raz

New instant message 23.10.2016 02:18
znovu to zkusim

New instant message 23.10.2016 02:13
tak to zkusmeeee

New instant message 23.10.2016 02:05
423 fg fgfgifngf

New instant message 23.10.2016 02:03
123 fg fgfgifngf
gfg fgf gf gfgflgfmgilfgfgfgf gf

New instant message 23.10.2016 02:03
123 fg fgfgifngf
gfg fgf gf gfgflgfmgilfgfgfgf gf
g fgfg fgf
123 fg fgfgifngf
gfg fgf gf gfgflgfmgilfgfgfgf gf
g fgfg fgf
123 fg fgfgifngf
gfg fgf gf gfgflgfmgilfgfgfgf gf
g fgfg fgf

Figure 6 Screenshot of android application activity views

4.3. Data storage

4.3.1. Shared preferences

“The SharedPreferences class provides a general framework that allows users to save and retrieve persistent key-value pairs of primitive data types. The users can use SharedPreferences to save any primitive data: booleans, floats, ints, longs, and strings. This data will persist across user sessions (even if the application is killed)” (Android Storage Options, 2016). Two different types of preferences can be used:

- preferences,
- shared preferences.

Basic preferences offer the possibility to use one file for every activity. This option therefore does not allow to share data across multiple activities compare to shared preferences which are identified by the preference name and which can be shared across multiple activities easily.

4.3.2. Internal storage

”You can save files directly on the device's internal storage. By default, files saved to the internal storage are private to your application and other applications cannot access them (nor can the user). When the user uninstalls your application, these files are removed.” (Android Storage Options, 2016)

The internal storage can be accessed by calling openFileOutput or openFileInput methods with a desirable file name to return the file stream.

4.3.3. External storage

“This can be a removable storage media (such as an SD card) or an internal (non-removable) storage. Files saved to the external storage are world-readable and can be modified by the user when they enable USB mass storage to transfer files on a computer.” (Android Storage Options, 2016)

This device does not need to be available all the time, but it is more suitable for a bigger amount of data than the Internal storage. It is suitable when data needs to be shared with other application. It also requires additional permission to the application granted by the user. This permission is declared in the manifest.xml file as shown in the following code block.

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

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

Where WRITE_EXTERNAL_STORAGE implicitly declare READ_EXTERNAL_STORAGE, therefore, always only one of these needs to be declared.

4.3.4. Database

Android offers a possibility to use SQLite database within the application. The database created by the application is usable throughout the application but not outside of it. This storage is suitable for a larger amount of data, especially if the data are structured or the data need to be queried.

4.3.5. Storages used in the application

There are several data which needed to be stored permanently within the application. In this section, it is described what were the cases and what storage is used for the given purpose.

Authentication token

The first case was an authentication token when the user logged in with his device to the web service. The authentication token is a simple string. It needs to be stored throughout the application and it needs to stay private. The suitable solution here was to use the internal storage and save it as a key value as shown in the following code block.

```
public void putBearerToken(String token) {
    SharedPreferences sharedPreferences=
        context.getSharedPreferences(RESOURCE_FILE_NAME, 0);
    SharedPreferences.Editor sharedPreferencesEditor =
        sharedPreferences.edit();
    sharedPreferencesEditor.putString(KEY_AUTH_TOKEN, token);
    sharedPreferencesEditor.commit();
}
```

Note that RESOURCE_FILE_NAME and KEY_AUTH_TOKEN are constants of the class. Retrieving the token is then very straightforward as shown below.

```
public String getBearerToken() {
    SharedPreferences sharedPreferences =
        context.getSharedPreferences(RESOURCE_FILE_NAME, 0);
    return sharedPreferences.getString(KEY_AUTH_TOKEN, null);
}
```

Notification data

Another case when data needed to be stored permanently was the notification data itself. The text data are not too large and it was better to keep them private so the decision was to use the internal storage. As a data format was selected JSON. A notification object structure is shown in the code block which follows.

```
{
    id: String,
    dateTime: long,
    title: String,
    text: String
}
```

A complete data object is an array of these JSON objects serialized and stored in a file. When only a part of the data needs to be read, the complete file is loaded into memory, parsed and the needed data are extracted. While inserting another data the complete file needs to be read and deserialized as well, the data are extended with another JSON object being pushed into the array, serialized and written back to the file.

5. Implementing notifications

5.1. Firebase Cloud Messaging

Notifications in an Android device can come from two different sources – the application itself can generate the notifications on the phone – that means the notification comes from inside the phone. Another possibility is to use some service to send push notifications to the phone. To implement sending push notifications from outside, Google offers two platforms: Google Cloud Messaging and newer Firebase Cloud Messaging.

Firebase is a platform made by Google to support the development of the Android and iOS applications. It offers a variety of tools to which the most important belong database and storages, authentication options, cloud messaging, hosting, crash reporting, advertising.

Firebase Cloud Messaging is one of the services provided which allows delivering notifications to the user device. Supported devices are iOS, Android and Web browsers. A part of the service is access to the Notifications Console GUI which is a web interface to send the notifications directly from Firebase website. It is the easiest way how to start sending notifications manually. Another option is to integrate one's own service through the HTTP or XMPP interface. The illustration of this can be seen in Figure 7.

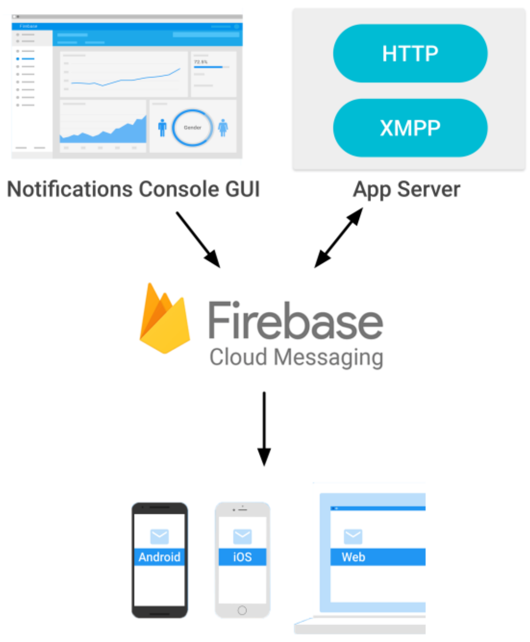


Figure 7 Firebase Cloud Messaging structure (Cloud messaging, 2016)

5.2. Notification targets

5.2.1. User segment

A segment can be defined as a number of users which the notification is being sent to. The segment is specified by the application name, language and application version. Not all these attributes are mandatory, and the notification can be sent, e.g. to all the users of the application. This type of notification is, therefore, applicable when the content should be delivered to a majority or all the users of the application.

5.2.2. Topic subscribers

This option is used to send notifications to a group of users not depending on the application or device settings but on the groups of users the device belongs to.

To subscribe and unsubscribe to different topics the client code is very simple as shown in the following code block.

```
FirebaseMessaging.getInstance().subscribeToTopic("news");
FirebaseMessaging.getInstance().unubscribeFromTopic("news");
```

If the topic exists, the device will be part of the user subscribed into the group, if the topic does not exist yet, it will be created automatically.

5.2.3. Device groups

Another option is targeting the device groups by creating and managing the groups by oneself via the service HTTP interface. An example request performing creation of a device group is shown in the following code block. (Device Group Messaging on Android, 2016)

```
https://android.googleapis.com/gcm/notification
Content-Type:application/json
Authorization:key=API_KEY

{
   "operation": "create",
   "notification_key_name": "appUser-Chris",
   "registration_ids": ["4", "8", "15", "16", "23", "42"]
}
```

The registration_id is the device token and notification_key_name is the group segment identifier used to send the notifications to the particular group of users.

5.2.4. Single user

The last possibility is to target a single user device with the notification. To be able to send the notification to the user, a firebase registration id needs to be acquired from the device. The id uniquely identifies the device for the Firebase Cloud Messaging so the service knows to which device to send the message to.

5.3. Notification types

5.3.1. Notification message

The notification message is automatically displayed to the end user on the device when it arrives. It has a size limit 2KB and needs to contain JSON object in the following format.

```
{
    "to": "firebase_token",
    "notification" : {
        "title": "Title of notification",
        "body" : "Notification text",
    }
}
```

It can optionally contain also ”data” JSON object which is delivered to the application when the notification is clicked. If using this type of notification, the developer has fewer options about operations which occur when the notification arrives into the device, however, it is easier to implement.

5.3.2. Data message

Data message is not directly a notification to the mobile device. This type of message will not create any notification when it arrives. Instead of that, it will trigger a method onMessageReceived of the FirebaseMessagingService class. The request structure to the Firebase service is shown below.

```
{
    "to": "firebase_token",
    "data" : {
        "key": "value",
        "key": "value",
    }
}
```

Since the notification is not created by the service itself, this service can be used to implement any kind of behavior the developer wants to trigger inside of the mentioned method. The method receives an object of RemoteMessage containing all the key-pair data specified in the request which could be in size up to 4KB. (About FCM Messages, 2016)

5.4. Sending text notification

The workflow of sending the notification from the web interface into the Android mobile device is described in the following nine steps.

1. Front end web application sends authenticated request to the web service containing the content of the notification – device ID, notification text.
2. Web service gets the username from the request and resolves the Firebase token to which it should send the notification by the device id
3. Web service stores the content in a MongoDB database
4. Web service fires the asynchronous single-user request for data message notification to the HTTP server using the Request Node.js plugin. The request is constructed with the code which follows.

```
const
    request = require('request');

module.exports.sendNotification = (id, firebaseToken, title, body) => {
    var body = {
        to: firebaseToken,
        data: {
            id: id,
            title: title,
            text: body
        }
    };
    var options = {
        uri: 'https://fcm.googleapis.com/fcm/send',
        method: 'POST',
        headers: {
            'Authorization': 'key=' + process.env.FIREBASE_SECRET_TOKEN,
            'Content-Type': 'application/json'
        },
        json: body
    };


    request(options, function (err, res, body) {
        if (err) {
            console.log(err);
        };
    });
}
```

5. The message arrives at the mobile device and triggers the overwritten method onMessageReceived(RemoteMessage remoteMessage) of the Firebase Message Service.
6. The data from the message are extracted and stored in a private file in the Android Local Storage.
7. Custom notification for the user is constructed and shown to the user.
8. If the application is running itself, the custom code also sends an intent to the main application to notify that the data have been changed and the current view needs to be rendered again with the new data.
9. When the user clicks the notification, the Detail Activity view will be opened with the content of the message.

This workflow ensures that the text content is downloaded from a web server immediately when the data arrives at the device and those are stored in the Local Storage. No internet connection is needed after the content is received to display the content.

6. File transfer

6.1. File storage design

A very important decision to make was where to store the files which the user would like to access from his phone. The simplest solution would be to upload files directly to the web server by using the HTTP protocol which unfortunately brings several disadvantages. At first, the web service is not designed to be a file storage, storing files inside a folder of the web server would be against all good practices how to build a web service. The storage would not be anyhow separated from the code and files would be available only through the web service which would increase the required resources. Another disadvantage is that files would be removed after every deployment and the storage itself would be very limited.

An alternative solution was needed where to store the uploaded files. This alternative was to use a separate service where the files can be stored. Because the application already uses a great deal of services from Amazon Web Service, it was desired to keep everything inside the same cloud solution and store the files in Service S3 which is already used as a storage for the front-end application.

6.2. Using S3 storage

To be able to use S3 service, first a bucket with a unique name and a region needs to be created. A bucket was created with a name “instant-mobile-receiver-uploaded-files-test” in the region “eu-west-1”. The bucket is private and can be only accessed by the AWS user with correct access policies.

6.3. S3 security

Two options how to upload files into S3 service are CLI application and S3 web interface. Those are already described in chapter 3.3.2. These solutions require either running a command line or manually uploading the files through the web browser. This is not suitable because the application needs to be able to access the S3 storage through some exposed interface. Another possibility to access the S3 service is by using HTTP protocol for which Amazon offers an SDK for JavaScript in the Browser.

This is a good and relatively easy option how to implement client with access to S3 service. The problem is that the S3 bucket needs to be either publicly accessible or the client code must contain AWS credentials to be able to access it. Neither of those options is good for this application since the files cannot be publicly accessible. An even worse solution would be to insert AWS credentials inside the client code. The AWS credentials consist of AWSAccessKeyId and AWSSecretKey. With these credentials client could have, depending on the access policies, access to some or all AWS services. AWS allows to create a user who would have, e.g. access only to the S3 service or even to only a single bucket of S3, but that would still mean that any user with the credentials could see and manage files from different users which would be an obvious security risk. Solution to the authentication problem was to use some authorization layer which will grant the right permissions for every request.

6.4. API Gateway

6.4.1. About the service

“Amazon API Gateway is a fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale. With a few clicks in the AWS Management Console, you can create an API that acts as a “front door” for applications to access data, business logic, or functionality from your back-end services, such as workloads running on Amazon Elastic Compute Cloud (Amazon EC2), code running on AWS Lambda, or any Web application.”(Amazon API Gateway, 2016)

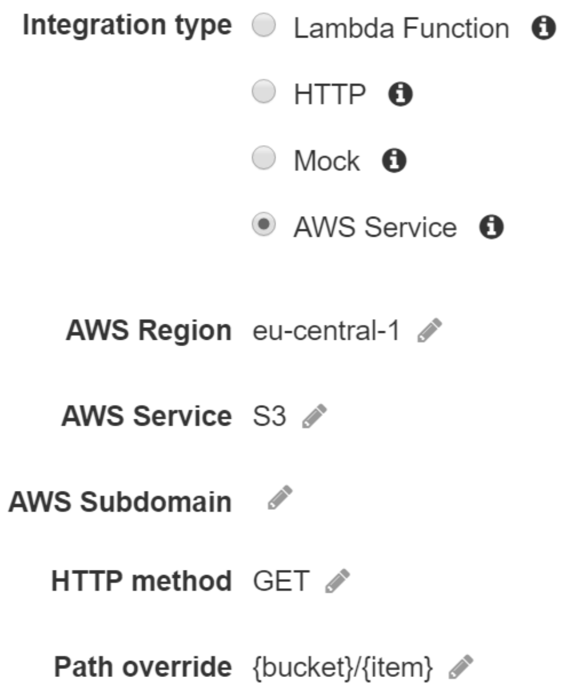
6.4.2. Creating API

It is sufficient to provide only the name of the API to create one. API created for the application is called “instant-mobile-receiver-file-api”. Every API consists of resources which are mapped to the requests coming into the API. Resource names can be either static or use parameters. For example, resource `/test` handles request which comes to the path ’/test’. To create a parametric resource in API Gateway, the developer uses the notation with parentheses, e.g. resource `/{test}` which handles all the requests starting with a slash. The value in the place of the ”test” string is mapped as a parameter.

Resources as such do not handle any requests. To handle the requests some methods need to be created under the resource. HTTP methods are GET, PUT, POST, HEAD, DELETE, OPTIONS, PATCH. While creating the method, API offers four integration types:

- lambda function – executes lambda function within AWS,
- HTTP – connects request to any HTTP service,
- Mock – does not connect request to any external service, the API itself creates a response depending on the mapping,
- AWS Service – creates integration with another AWS service.

For the purpose of this application, the suitable option is the AWS Service integration type. While setting up the method, the correct service and region need to be specified. Correct AWS Service is S3 and corresponding AWS Region is "eu-west-1" where the bucket is created. The path has two parameters "bucket" and "item" which are mapped to key-value pairs item and bucket. These values are parsed from the URL path and passed to the S3 service as a body of the request. The settings of one of the methods can be seen in Figure 8.



Integration type	Lambda Function HTTP Mock [✓ AWS Service]
AWS Region	eu-central-1
AWS Service	S3
AWS Subdomain	
HTTP method	GET
Path override	{bucket}/{item}

Figure 8 API GET method settings

6.4.3. Granting permissions to API Gateway

API Gateway has only those access rights which are granted to it by its settings. It cannot access any resources of the AWS until it has granted the right permissions. It was needed to grant the API permission to perform certain operations on the S3 service, more precisely on the bucket called “instant-mobile-receiver-uploaded-files-test”.

AWS using documents in JSON format to create policies. The schema and grammar are described in the documentation. Permissions for the API are shown below.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "s3:*",
            "Resource": ["arn:aws:s3:::instant-mobile-receiver-uploaded-
files-test/*"]
        }
    ]
}
```

There are two important things to notice - first is the action value which is specified as “s3:*” which states that all the actions started with “s3:” can be executed. These includes e.g. s3:GetObject, s3:PutObject and many others. The second important feature is the resource value, which specifies what resources can be accessed. This policy allows access to any file in the “instant-mobile-receiver-uploaded-files-test” bucket within the S3 service. The character “*” stands as a wild character for any string. The policy which is created needs to be the first assign to the role which is then assigned to the single methods in the API.

6.4.4. User authentication with lambda function

The API is set up that all the users have access to the same resources. This is still not suitable because users could access resources from other users. Another user-specific authentication for the API is therefore needed. This is possible by changing authorization settings of the methods of the API. To create custom authentication, the method needs to have assigned a custom authorizer. The authorizer is a lambda function inside the AWS.

”AWS Lambda is a serverless compute service that runs your code in response to events and automatically manages the underlying compute resources for you. You can use AWS Lambda to extend other AWS services with custom logic, or create your own back-end services that operate at AWS scale, performance, and security.” (AWS Lambda Details, 2016)

When the lambda function is created the authorizer must be created in the API and the lambda function is assigned to it together with the Identity token resource attribute which specifies what part of the HTTP request will be mapped to the authorization token inside the lambda function. The authorizer’s Identity token source, in this case, is ”method.request.header.AuthKey” which specifies that the requests need to have AuthKey header with the correct authorization token. The value of AuthKey header is then accessible inside the lambda function.

Lambda function for this authorizer uses exactly the same authorization mechanism as the web service. The validation of the token is executed the way shown in the following code block (content of the methods is not shown for simplicity).

```
exports.handler = (event, context, callback) => {
  // has the token value
  let token = event.authorizationToken;
  // access the database
  this.findEmailFromToken(token)

  .then(email => {
    // constructing policies
    let policy = this.constructPolicy(email);
    callback(null, policy);
  })

  .catch(err => {
    callback(null, err);
  });
}
```

The method findEmailFromToken queries the database and resolves the token to the email of the user which tries to perform the operation. If the token is successfully resolved into an email, the function creates access policies for the user and returns it back to the API.

The important thing to distinguish is that the access policies which are constructed are access policies to the API Gateway. Policies described in subchapter 6.4.3 are access policies for the API Gateway to have access to the S3 service. On the other hand, policies returned from the lambda function are policies for the user to access the API method itself. The return policies are shown in the below code block.

```
    "principalId": "1",
    "policyDocument": {
        "Version": "2016-10-29",
        "Statement": [
          {
            "Action": "execute-api:Invoke",
            "Effect": "Allow",
            "Resource": [
              "arn:aws:execute-api:eu-west-
1:732042213639:r5wc3eim09/test/GET/instant-mobile-receiver-uploaded-files-
test/" + <escaped-email> + "%2F*",
             "arn:aws:execute-api:eu-west-
1:732042213639:r5wc3eim09/test/PUT/instant-mobile-receiver-uploaded-files-
test/" + escape(email) + "%2F*"
            ]
          }
        ]
    }
```

The important part of the policies is again the resource values. They allow the user to execute GET and PUT requests inside the API with the path having the bucket value ”instant-mobile-receiver-uploaded-files-test” and item value <escaped-email>%2F*. Since the values are mapped to the S3 service as a name of the bucket and the name of the file, the users can only access items which path starts with the value of its own email address. That means they can only access files inside a folder named by their own email address.

6.5. Accessing files

The API is configured to expose methods GET, PUT and OPTIONS of the S3 bucket allowing uploading and download the files. It has also configured the CORS headers to be able to access the resources from any website.

Now accessing the resource is simple through HTTP requests authorized with the user token in the AuthKey header. Below is an example of a code performing file upload.

```
$http({
        url: url,
        method: "PUT",
        data: file,
        headers: {
                'Content-Type': type,
                'AuthKey': authKey
        }
}).success(resolve)
  .error(reject);
```

6.6. Sending text file workflow

The workflow of sending text file content into the application is described in the following six steps.

1. A user drops a text file or chooses a text file from his computer to be uploaded into his mobile.
2. The frontend application uploads the file to the S3 service using the API Gateway with the correct authorization token. The file is uploaded to the folder with user email and the file is named with the current timestamp.
3. Frontend application sends the request to the web service with the URL of the uploaded file and other file information.
4. Web service inserts the record into the database, the example record is shown below.

```
{
  "deviceId": "51115591",
  "email": "luk@test.cz",
  "fileName": "test.txt",
  "fileUrl": "https://r5wc3eim09.execute-api.eu-west-
1.amazonaws.com/test/instant-mobile-receiver-uploaded-files-
test/luk@test.cz%2F20161030T122951",
  "firebaseToken": "21213311",
  "id": "1477830592090_luk@test.cz_40297",
  "type": "textfile"
}
```

5. Web service download the content of the text file and sent it to the mobile device.
6. The workflow continues the same way as simple text notification as described from fifth step onwards in chapter 5.4.

7. Results

7.1. Evaluation of requirements

The aim of the thesis was to develop a prototype of three separate applications which together provide a solution to deliver content from web interface to the mobile application. The first version of the application was developed. It allows the user to transfer plain text and simple text files into the device. If the user has multiple devices, it can choose to which of those the content will be delivered. The web interface also has login and registration features available. The static front end application was deployed to the S3 bucket in Amazon Web Services cloud.

The mobile application also has a login feature, which means the device is paired to the user account. The application is able to receive the content and display notifications and it is able to store the information and display it later.

The web service which the web and mobile applications are using is deployed to Amazon Web Services cloud in a Docker container so it can be accessed by the users.

Web service is secure in the meaning that it uses authorization and authentication for all the communications from client to the server. The reliability is ensured by using the Amazon cloud. Also scalability is guaranteed by using the cloud solution. AWS Service EC2 can be configured for higher computation power if needed or even to use auto-scaling to create cluster of servers. DynamoDB resource configuration is also possible to set up. Using more resources for both services brings more costs to running the project.

7.2. Future development plans

None of the applications is ready to be put to the production because of many features which could still be implemented so this would become a final and wide-usable service. New features to consider would be e.g. supporting more file types, creating an iOS application or implementation of sending content to all devices at once.

7.3. Code availability

The completely open source code is available in four separate repositories on the writer’s GitHub account in the following links:

- Web service: https://github.com/luke07/instant-mobile-receiver-api
- Web application: https://github.com/luke07/instant-mobile-receiver-frontend
- Android application: https://github.com/luke07/instant-mobile-receiver-android
- Lambda functions: https://github.com/luke07/instant-mobile-receiver-lambda-functions

8. Conclusion

While developing the application in the scope of this thesis I focused on learning new things which I was interested in, especially the cloud solutions. It gave me much freedom to experiment with various technologies, platforms and services and therefore helped me to increase my skills in modern web development which I am interested in at the moment.

The main idea here was to use the most recent, modern and popular technologies which are getting used by more and more companies in the world. Knowing these technologies has gotten me skills which are required from every good web developer. Namely, Angular.js, Restify framework, DynamoDB, Firebase, Docker, Amazon Web Services were all totally new technologies for me which I found very interesting and which I will hopefully use in the near future. The AWS services used within these applications are Lambda, DynamoDB, API Gateway, Elastic Beanstalk, EC2, S3, IAM for creating policies, Code pipeline for deploying and CloudWatch for logging. Other tools used from AWS are AWS CLI tools and AWS SDKs for Node.js.

The last but not least interesting thing I learned is the importance of a good architecture design when working with separate applications which need to communicate and work together. Many times during a development I needed to stop programming and re-think and redesign slightly some part of the communication process between the application. Later I started to develop very small features one at a time and integrate them throughout the applications. This taught me to use an iterative development process even when working alone and not as part of a team.

9. References

About FCM Messages. A page on the Firebase service. Accessed on 1.11.2016. Retrieved from https://firebase.google.com/docs/cloud-messaging/concept-options

Amazon API Gateway. A pppage on the official AWS documentation. Accessed on 4.11.2016. Retrieved from https://aws.amazon.com/api-gateway/

Android Storage Options. A page on the official Android documentation. Accessed on 15.10.2016. Retrieved from https://developer.android.com/guide/topics/data/data-storage.html

AWS Lambda Details. A page on the official AWS documentation. Accessed on 28.10.2016. Retrieved from https://aws.amazon.com/lambda/details/

Cloud messaging. A page on the Firebase service. Accessed on 6.11.2016. Retrieved from https://firebase.google.com/docs/cloud-messaging/

Device Group Messaging on Android. A page on the Firebase service. Accessed on 13.10.2016. Retrieved from https://firebase.google.com/docs/cloud-messaging/android/device-group

Fielding, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*, Doctoral dissertation, University of California, Irvine, 2000.

Mark Masse, *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*, O'Reilly Media, Inc., 2011.

Provisioned throughput. A page on the official AWS documentation. Accessed on 2.10.2016. Retrieved from http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ProvisionedThroughput.html

What is AWS Elastic Beanstalk. A page on the official AWS documentation. Accessed on 14.10.2016. Retrieved from http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/Welcome.html

What is Docker. A page on official Docker website. Accessed on 10.10.2015. Retrieved from https://www.docker.com/what-docker