Providing Secure Web Services for Mobile Applications

Tero Kivisaari

Helsinki 12.3.2015
M. Sc. Thesis
UNIVERSITY OF HELSINKI
Department of Computer Science
Providing Secure Web Services for Mobile Applications

Changing consumer behavior drives the demand for convenient and easy-to-use mobile applications across industries. This also impacts the financial sector. Banks are eager to offer their services as mobile applications to match the modern consumer needs. The mobile applications are not independently able to provide the required functionality; they interact with the existing core business functions by consuming secure Web Services over the Internet.

The thesis analyses the problem of how a bank can enable a new secure distribution and communication channel via the mobile applications. This new channel must be able to interact with existing core systems. The problem is investigated from different axis related to Web Services protocols suitable for mobile use, security solutions for the communication protocols and the required support available in the selected mobile operating systems.

The result of the analysis is an architectural description to fulfill the presented requirements. In addition to constructing the architecture, the thesis also describes some of the more advanced threats targeted against mobile apps and Web Services and provides mitigation schemes for the threats. The selected architecture contains a modular security solution that can be utilized outside of the financial context as well.

ACM Computing Classification System (CCS 2012):
- Information systems → Web Services
- Security and privacy → Software and application security
- Software and its engineering → Software architectures
## Contents

1. Introduction
2. Background and requirements for secure Web Services
   2.1 Native mobile applications
   2.2 Stateful to stateless services
   2.3 Basic security principles
   2.4 Mobile security and general threats to be mitigated
   2.5 Relevant legislation and regulation
   2.6 Other constraints and requirements
3. Selecting the appropriate type of Web Services for mobile consumers
   3.1 SOAP based Web Services
   3.2 RESTful Web Services
      3.2.1 XML based RESTful Web Services
      3.2.2 JSON based RESTful Web Services
   3.3 Comparing the features of SOAP based and RESTful Web Services
   3.4 Recommended pattern for Mobile use
4. Security features of SOAP based and RESTful Web Services
   4.1 Security of SOAP based Web Services
   4.2 Security of XML based RESTful Web Services
   4.3 Security of JSON based RESTful Web Services
   4.4 Conclusion and recommended security solution
   4.5 Web Services related security considerations
5. Remote communication and security feature support in mobile platforms
   5.1 Apple iOS
   5.2 Google Android
   5.3 Microsoft Windows Phone
   5.4 Common denominator in supported remote communication protocols
   5.5 Other security related considerations for mobile platforms
6. Suggested solution for offering secure Web Services for Mobile Applications
   6.1 Architectural description of the solution
      6.1.1 Conceptual architecture
      6.1.2 Mobile Application Architecture
      6.1.3 REST API Architecture
      6.1.4 Physical architecture
   6.2 Mobile device and mobile app registration
   6.3 Token based security solution for session management
1 Introduction

Financial institutions, such as banks, are eager to offer their services in ways that match the modern consumer expectations and provide the maximum convenience for the users. One recent major trend is offering native mobile applications that can be downloaded from the online application stores and installed on the end users smart phone and tablet devices. A common requirement for these applications is secure invocation of remote services for interaction with existing company core business functions. In terms of communication protocols, messaging formats and security solutions there are several possibilities how to arrange the remote communications. This thesis investigates some of the alternatives that have reached sufficient maturity for a large financial corporation (hereby referred as the Bank) and presents an architectural solution for building a new mobile communication and distribution channel. This new channel will enable access to the same core business functions which can currently be reached via the Bank’s existing channels, such as Internet Banking portals and Contact Centers.

The Bank has chosen Service Oriented Architecture (SOA) as one of the primary patterns to manage, control and utilize the large amount of internal services. Existing and still developing SOA architecture inside the Bank is the foundation and one of the driving forces influencing how to develop and design future services that are exposed to the open internet for mobile consumers. One typical property of SOA services is that they are stateless [Erl06]; stateless services bring several benefits with simplified service design and excellent scaling characteristics. These benefits will specifically realize in multilayered applications [RiR07].

Providing stateless services for consumers outside of the company’s internal network barrier also introduces new challenges in addition to the mentioned benefits. These challenges commonly relate to how to manage context data related to the user session, and especially, how to manage the authentication state of the user. If a service is stateless, session state is not hold by the service provider. This means that every request must contain all the necessary context data for the service to be able to process the request including the authentication related data. Independent authenticating of request messages is an obvious change in paradigm when comparing to the traditional secure Web applications where the authentication state is stored by the server after the initial user authentication.
Simple Object Access Protocol (SOAP) based Web Services on top of HTTP protocol have for years been the de-facto technical solution for publishing and invoking autonomous services in internal SOA architectures [CCY11]. SOAP protocol as such is not perfectly matched for mobile use for various reasons detailed later in the thesis. The main challenger for SOAP is Representational State Transfer (REST, or often “RESTful”) based services that have quickly gained a de-facto status among Web developers and surpassed SOAP based services in the internet facing usage [VVP12]. A key promise of REST is to be simple and adhere to the basic structures that are built in to the World Wide Web [Fie00]. SOAP based Web Services and RESTful services fulfill to the same fundamental client-server pattern, but also differ in many ways such as transport protocols, message content types and security.

To understand which messaging protocol is best suited for the selected mobile platforms; Google Android, Apple iOS and Microsoft Windows Phone, it is important to acknowledge what kind capabilities these platforms possess in terms of remote communication protocols and security features. Current mobile platforms and devices are lacking some of the processing power, network bandwidth and the mature authentication standards that are available with traditional computing platforms. There is a strong need to come up with a creative solution that will still conform to the fundamental security principles and strict requirements of the financial institutions.

Using the constructive research method this thesis produces an implementation-independent architectural description for a mobile channel to access and consume financial services in a secure way. Input for the construction comes from Web Service based remote communication protocols and security solutions on top of those, mobile platform support for remote communications, common internet related threats and generic requirements and constraints set by the Bank. These wide topics are addressed with a rather high abstraction level with occasional deep dives in to the most relevant areas. Decision to describe the architecture in an implementation independent level is done to maintain an abstraction level that enables to results to be reused in various heterogeneous environments. These might include for example public or private cloud, physical or virtual on premises servers depending on the concrete requirements and existing infrastructure used within the implementing organization.
Structure of the thesis is as follows. Chapter 2 presents the general landscape around native mobile applications and Web Services targeted for mobile use. Chapter 2 also explains basic security principles and describes general requirements for the solution, taking into account relevant regulation, legislation and constraints inherited from the existing architectural landscape in the Bank. Chapter 3 introduces and compares the remote communications protocols and analyses their strengths and weaknesses for mobile usage, while Chapter 4 focuses on the security properties of these protocols. Chapter 5 introduces the targeted mobile platforms which are going to be the service consumers and describes their capabilities related to remote communications and local security features provided by the platform. Finally Chapter 6 is the construction, it describes a logical architecture for providing the secure Web Services for Mobile Applications based on the information presented in the previous chapters and maps the concerns and requirements presented throughout the thesis into the final solution. The concluding chapter provides a summary explaining why RESTful services with JSON have been selected as the integration protocol and highlights the key building blocks of the security solution.

2 Background and requirements for secure Web Services

Smart phones are coming more and more popular; already in 2012 70% of the mobile phones sold in Finland have been smart phones [MVO12] and today the number is most likely even higher although recent statistics are not available. One of the key attractions of the smart phones are native mobile applications that are often tailored to provide a bit more limited set of functionality when compared to traditional Web applications but in a very user friendly and convenient way. The relatively small displays of the devices sets constraints to much functionality can be embedded to the screens.

Obviously also financial institutions are eager to be able offer their services in a way which maximizes the user experience and convenience without tying the user to specific location or time to access the services. In the modern information society it is simply not enough for corporations to dictate in which channels their services will be available, but they must be able to understand and adapt to the preferred ways in which the customers wish to interact with the services the corporations provide.

In the context of this thesis the purpose of the mobile applications is to provide a new
channel to the Bank. Customers are able interact with the same business processes which could be alternatively reached via the existing channels such as the Internet banking portals, phone based contact centers or via the physical branch offices. The native applications themselves can not independently establish a channel to the bank. For the apps to be able to communicate to the bank some kind of online service interface is required. This service interface is the “enabler” for the mobile or any other types of applications running outside of the Bank.

Communications between the mobile app and the Bank online service interface are the most sensitive parts to be protected. Security of the mobile platforms and apps running on those platforms is managed by the mobile operating systems leaving limited room for the application developers themselves to enhance security.

Financial services provided by the Bank obviously need to be very well protected to maintain the trust of the consumers and also protect the Bank and the customers from the malicious attackers. The other way around, the Bank must also be able to trust that the customers are who they claim to be. In order to fulfill this requirement, strong electronic authentication is needed. The Bank operates in several European countries. All of the countries have different local authentication solutions. Some are directly provided by the Bank, some might be national “sector” solutions covering multiple financial institutions, governmental and commercial services and some of the authentication solutions are provided by third party service providers. Since the aim is to have a common online service interface for multiple countries it must implement a pluggable authentication solution which enables the use of any of the local user authentication solutions without impacting the actual business services.

Cybercrime in the internet is growing with an incredible speed as identified by Software Security provider McAfee [McA14]. Criminals are always keen to follow the money trail; wherever there is a possibility to earn money they are willing to plan cunning attacks. One good example of recent attacks against the Finnish banks can be found from Tietoviikko magazine [Tiv12]. Tietoviikko investigates and explains a recent Man-In-The Browser (MITB) virus epidemic that was specially customized and targeted to attack the Finnish internet banking customers. Recent reports from F-Secure [FSe14] also indicate a significant growth for mobile viruses and trojans targeting especially the Google Android platform. All of this goes to show that having a high quality security solution is critical for a corporation that wishes to avoid direct losses and maintain a
good reputation among customers.

This chapter describes the overall landscape, clarifies terms and presents common requirements originating from multiple different sources. There are definite hard requirements originating from national and EU legislation, banking sector specific regulation, constraints imposed by the mobile environment. Basic security principles obviously also need to be adhered to. Functional requirements regarding specific services or use cases the application should support are left outside of the thesis except for strong authentication which is always necessary for identifying the customer.

2.1 Native mobile applications

Mobile applications (often referred as “mobile apps” or just “apps”) can be divided roughly in to three different categories based on their technological foundation. Native applications refer to applications developed in a specific programming language for a specific platform [GoS14]. Hybrid applications refer to applications where part of the application is platform specific code and part of the application is utilizing platform agnostic technologies such as HTML5 and JavaScript [GoS14]. Mobile web applications only use the Web server and Web browser to implement the application [McG11] just like traditional Web applications with the exception that the content is optimized for a smaller screen. Obviously there are significant differences in the capabilities and costs depending on the selected pattern.

Native applications provide all the flexibility and performance the mobile platform has built in [McG11]. Then again they need to be independently developed to all selected mobile platforms which increases the development and maintenance cost.

Hybrid applications enable the use of platform specific functionality but aim to provide the bulk of the application in a way that can be shared for all selected platforms (often by utilizing HTML5) [GoS14]. Hybrid approach will typically decrease the cost related to the applications but might not guarantee a similar level of user experience as the native approach.

Mobile Web applications only use the browser for implementing the user interface and usually one implementation covers all platforms. This is a bit simplified picture, as the browser and JavaScript implementations differ between the platforms [McG11]. Mobile Web applications are the cheapest of the three. It is often possible to publish a new lay-
out for an existing Web site to make it mobile-friendly. However the use of platform specific functionality is limited and the user experience is typically the lowest of the approach. Marketing-wise mobile Web apps lack the visibility from the online application stores, which is available for native and hybrid applications.

Selecting the optimal pattern of mobile applications is not a simple task; various different aspects such as functional and non-functional requirements for the applications, capabilities of the organization and many others need to be balanced to find the appropriate strategy. In this context the decision has been done previously by the Bank. The strategy is to develop native applications and that is the frame for the rest of the thesis.

Situation with the selected mobile operating systems Apple iOS, Google Android and Microsoft Windows Phone is such that they all differ in regards to the supported programming languages, frameworks, integration protocols and supported security features. This may seem like a very familiar situation for people who have been involved in developing distributed systems over the past decades. CORBA architecture and then more recently SOAP based Web Services have tried to provide an answer to the same exact problem; how to enable platform and programming language independent interoperability for distributed systems.

In order to avoid building separate Web Service interfaces per mobile platform or implementing proprietary communication and security solutions, it is important to find the common nominators in terms of the supported communication protocols and messaging patterns. Selected mobile platforms and the common nominators are explained more detail in Chapter 5.

### 2.2 Stateful to stateless services

When transitioning from traditional Web applications in to native mobile applications it is possible to start building systems which utilize stateless Web Services. Stateless services in general simplify development and testing of services and enable the building of more reliable systems [RiR07] by limiting the scope of an individual service. Stateless services also bring significant scalability improvements as there is no session state to be stored, maintained and synchronized in the server side within the Web layer. This enables the Web layer to scale horizontally [Fie00]. The Bank has a substantial user base and scalability is an important requirement.
It is essential to understand that the concept of stateless services does not mean that there is no state; it means that the state is not held within the Web applications. In a stateless service architecture state is typically stored within the actual core applications in the data they hold. Session related state is held by the client application. Similarly the state could be managed by a Business Process Engine (BPE). BPE is responsible for orchestrating long running business processes where preservation of the process state is crucial, but the Web Services that are used to access the BPE can are still not required to hold an intermediate state.

For a service to be stateless all information required for processing an individual request must be delivered within the request message. This means that all messages can be processed individually without any knowledge of the previous message exchange. It also means that no activities need to be coordinated between clustered servers, no state information is shared and no session affinity is needed [RiR07]. Figure 2.2 illustrates how the session context has been moved from server side to the client side. Moving the session context into the client application effectively simplifies the needed architecture for server side session management. At the same time the move also introduces new requirements for the client applications and Web Services which will be further detailed in the following chapters.

Figure 2.2 Simplified architecture of stateful and stateless services. On top of the picture is a traditional web application and on the bottom a native mobile application that utilizes stateless services. User’s session context is marked with a yellow star
In addition to the mentioned benefits, stateless services also provide new challenges. If all of the information needed for processing an individual request must be delivered within every message, message size will increase [Fie00]. Another challenge is related to request authentication; if each request has to be processed independently of the previous requests, it means that the authentication data must be presented with every single request.

If this model is compared to how Finnish banks have commonly implemented their strong electronic authentication systems with the One Time Code cards (OTC) there is a clear gap. In the current OTC based solutions a successful user authentication generates a server side session and a session identifier referenced in a cookie. The cookie is subsequently attached to the request message to prove that the request is part of an authenticated session held by the server. To enable stateless services there is a clear need to be able to replace the cookie with a self-containing solution that can independently contain the authentication state.

Existing One Time Code based authentication solutions are currently the corner stone of many Finnish services requiring strong electronic authentication (such as Finnish Social Insurance Institution, Finnish Tax Administration, Internet Banking portals). Other solutions such as electronic ID cards and Mobile Certificate exist but are not widely used in Finland [Jut14]. In other Nordic countries solutions such as the Mobile Bank ID¹ and hardware tokens have gained wide popularity [FIT13]. Some creative thinking is needed to come up with a new solution which can facilitate the use existing OTC infrastructure as well as support multiple other possible strong authentication solutions and still gain the benefits of basing the solution in stateless Web Services.

### 2.3 Basic security principles

Basic security principles in distributed systems are not tied in to the actual implementation platforms whether it is mobile, desktop or Web applications. This section introduces some of the basic security principles in a more generic level. Mobile app specific threats are detailed further in the next sections.

Depending on the sensitivity of the data processed with the mobile applications there might be several different security requirements for how to arrange and protect the re-

---

¹ Mobile Bank ID is a mobile application based strong electronic authentication solution used in Sweden and Norway.
mote communications and end user applications. In this study these requirements commonly relate to (but are not limited to) verifying the identity of the end user by using strong electronic authentication, protecting the integrity and confidentiality of the messaging exchange, hardening the end user mobile applications and online Web Service interfaces.

One fundamental security concept is Authentication, Authorization and Accounting [PaS09] often referred as AAA. This concept is typically applied in almost all systems that are processing sensitive data and which need a mechanism for limiting the access to the system.

**Authentication** refers to verifying the identity of the requester of a service [Dec09]. The mechanism for authenticating the identity of a subject typically falls in to one of three categories. In the context of electronic authentication these categories are often referred as factors. The number of factors used for authenticating corresponds how strong or reliable the authentication is.

The possible factors are [Fin14]:

1. Something that only you know. This can be a user-id, password, PIN code or some other information that only you should know.
2. Something that only you possess. This can be a credit card, an OTC list, a hardware security token, an electronic id card or even a previously registered mobile device (this is elaborated further in Chapter 6).
3. Something unique that you are. Finger prints, iris, DNA, voice or key board tapping pattern for example.

When two or more authentication factors are combined within the authentication process, the authentication mechanism can be considered to belong to the category of strong electronic authentication [Fin14].

**Authorization** refers to the process of verifying that the requestor of a service has the permission to consume the requested service [Dec09]. Authorization is performed after authentication after the identity of the requestor has been verified. There is also a need for default policy for the non-authenticated user. Default policy might simply state that all other services, except the authentication service, are disallowed for the non-authenticated user.
Accounting refers to an audit trail or a log file indicating the exact actions that have been taken by the customer [Dec09] and the outcomes of those actions. Detailed audit trail enables traversing back in time for example in the case of a dispute to clarify the exact steps and actions taken within a business process.

To illustrate what AAA means in practice, an example of a customer withdrawing money from an ATM machine with his credit/debit card is used. The process starts with authentication when the customer enters his credit card in to the ATM and enters the related PIN code. This is obviously two-factor authentication; something that the user possesses (the credit card) and something that the user knows (PIN code of the card).

Second step is authorization, it is not possible withdraw money from any account in the bank, but only from the accounts that you are authorized to use. To be more specific only from the account that is associated with the specific card being used. Authorization needs to happen with every single action be it withdrawal or just viewing the balance. Since it occurs behind the user interface it might feel too obvious even to think about. Authorization is often divided in to two categories, coarse grained and fine grained. Coarse grained authorization might refer to a less detailed list of which accounts the customer has in his/her control and fine grained authorization refers to the actions the customer is allowed to execute for a specific account. For example view only, deposit or withdraw money.

Accounting can not be seen as an individual step in the withdrawal process. It happens continuously in parallel with every single step that the customer takes starting from registering the card that has been entered to the ATM. When the process is completed the Bank will have a complete audit trail of the events that occurred. These events include the location where the customer used the ATM, the time when did he/she use it, which actions the customer took and what where the results of those actions. Final entry in the audit log describes when did the customer terminate the session at the ATM and picked out his card and the money. The bank will store this information for the coming years according to its accounting policy to be able to prove the exact steps which occurred in case there would ever be a dispute of how much money was withdrawn and by whom.

The described security principles also apply to services used by mobile applications. In addition to the principles mentioned above, there are extra requirements caused by the nature of open internet; most importantly message integrity and message confidentiality.
Message integrity means that the integrity of the messages is guaranteed and preserved [Lad11]. After the message author has created the message there is no way an attacker could alter the message in such a way that it would not be detectable for the message receiver. Typical means for implementing message integrity contain Hashed Message Authentication Codes (HMAC) with symmetric keys and PKI based digital signatures with asymmetric key pairs. Signatures and HMACs can be created various different cryptographic algorithms. HMACs and digital signatures are elaborated in more detail in the following chapters.

Message confidentiality means that the message content is not revealed to anyone else than to the sender and receiver [Lad11]. This can be achieved by encrypting the messaging exchange. Encryption can occur in the transport layer via the use of Transport Layer Security (TLS, often also referred as SSL) or in the messaging layer with the use of a of message level encryption scheme. Difference between message and transport level protection is elaborated further in Chapter 4.

In the context of security principles for mobile Web Services it is important to distinguish two authentication related concepts which closely relate to each other; user authentication and request authentication. User authentication refers to verifying the identity of the actual end user based on a set of credentials provided by the user. Request authentication refers to authenticating the individual request to verify that it belongs to an authenticated session. The authenticated session must have been initiated by the end user by performing user authentication. Certain Public Key Infrastructure (PKI) based security mechanisms such as digital signatures asymmetric keys or SSL mutual authentication combine user and request authentication. This means that the user actually provides his/her identity with every single request. Such an approach effectively removes the need for maintaining any kind of sessions from security perspective.

2.4 Mobile security and general threats to be mitigated

As with any type of applications which are targeted for public use over the internet, also the native mobile applications are under a myriad of different kind of threats. These threats are targeted to all layers of the overall infrastructure. Open Web Application Security Project (OWASP) has a dedicated section for mobile application security and they list the current top 10 risks.
OWASP top 10 Mobile risks [OWA14]:

M1: Weak Server Side Controls
M2: Insecure Data Storage
M3: Insufficient Transport Layer Protection
M4: Unintended Data Leakage
M5: Poor Authorization and Authentication
M6: Broken Cryptography
M7: Client Side Injection
M8: Security Decisions Via Untrusted Inputs
M9: Improper Session Handling
M10: Lack of Binary Protections

Further details of the individual risks can be found from the OWASP Web site. From the list it is obvious that top 10 entries contain risks of various types. The risks are related to how the actual mobile applications are implemented and secured, how the transport layer to the services is protected and also what kind of security controls are in place behind the Web Service interface. This thesis and the security solutions presented in later chapters are primarily focusing on the transport layer, message exchange and the server side controls. The actual native mobile applications and mobile platform security features are not the main focus of this document and only light weight introduction will be provided. Latest information regarding the constantly changing threat landscape can be found from the Internet security providers and cybercrime prevention companies’ blogs.

Recent sophisticated virus attacks such as the Stuxnet [Sym10] have efficiently illustrated that there is no such thing as an un-penetrable IT system. It is possible to penetrate every single system connected to the internet and even the ones which are not. If enough will and resources are put on to the task there will be a way in to any system. With the revelations from Edward Snowden [Gua13] concerns have risen towards some of the fundamental structures of World Wide Web. If governmental organizations have the possibility to tap in to all internet traffic it is only a matter of time before cybercriminals will start to have the similar capabilities.
Banks have received their share of fairly advanced attacks. There are for example already viruses targeted against specific banks which will contaminate user’s mobile phone when connected to the infected PC [FiR11]. After contaminating the device the attacker waits for the victim to log in to the internet banking portal to steal the customer’s session. Even if the bank has a risk engine\(^2\) that might request an “out of band” confirmation by SMS in case of a suspicious transaction, the virus in the mobile phone will hide the received SMS and answer behalf of the user to confirm the forged transaction [Tec12]. Just to show how clever the attackers are, there are several variations for this attack. In one of them a modified version of the trojan changes the customers phone number within the Internet banking portal so that the confirmation message is delivered to a device possessed by the attacker. Another version is a Man-In-The-Browser (MITB) attack where the attacker displays forged content in the browser. The customer is made to believe that the bank is requesting the customer to make a test confirmation to make sure that bank has the correct phone number. Actually the attacker has performed a financial transaction on the background and the user confirms real money to be transferred out of his account [IBM12].

Even the PKI based signature and encryption mechanisms, which are generally regarded as being very secure, have several vulnerabilities as identified by Patel, Mohandes et al. in their paper “Attacks on Web Services and mitigation schemes” [PMP10]. Furthermore, Open SSL that is one of the most popular Open Source SSL libraries used, was found to contain multiple critical security vulnerabilities [CVE14]. The code for Open SSL is open source and has been open for peer reviews for several years. All of this goes to show that it is crucial to be prepared for major changes in the threat landscape and to have solid mitigation plans to ensure business continuity.

The aim of an organization planning to develop mobile applications for financial transactions should not be to create a system which is impossible to penetrate. A more realistic approach is to evaluate the risk level of the services provided, define the risk tolerance of the company and decide to invest in to the security solutions accordingly. Another a bit more simplified view is to create a system which is significantly more difficult to penetrate than the other competing systems on the market. Most of the cyber-

---

\(^2\) Risk engines (also referred as “transaction risk scoring engines”) are commonly used by banks to analyze money bearing transactions to profile and score risk level of the transaction based on various properties. High risk transactions might be stopped or require an extra confirmation from the user.
criminals work like any enterprises; they are after return for their investment. If it takes five times more time to attack a certain online system than another, most of the focus will be on the system which requires the least time to penetrate. As identified by the security vendor McAfee, the criminals want to get maximum financial benefit with the least possible effort and minimal risk of getting caught [McA14].

It is good to remember that decreased usability is one of the common tradeoffs for increased security [BKS11]. The more security controls are built and authentication factors are used, the more cumbersome it will be for the user to use the system. This is especially relevant for the native applications. One of the driving forces for the “app revolution” has been the excellent usability and optimal user experience of the native applications when compared to traditional mobile Web applications.

The most important thing is to acknowledge and understand the risks and to find a balance between security and usability. In practice it means taking a calculated risk; as long as that risk is small enough and acknowledged and accepted there is nothing wrong with that.

### 2.5 Relevant legislation and regulation

Publishing mobile applications which use strong electronic authentication and offer financial services such as payments or card related functionality is regulated in several ways. There is EU wide legislation, local legislation, sector specific regulation from the central organizations and security standards from commercial card providers such as Visa and MasterCard. This section briefly introduces some of the related regulation which is relevant in the context of this paper.


The basic principle of the directive is the same that was described previously in Section 2.2. For an electronic authentication method to be considered strong it has to utilize at least two out of three identification requirements (factors) as explained in Section 2.2. The law also makes a distinction between an “electronic signature” and a “developed electronic signature”. First one refers to generic strong authentication solutions for example with traditional one time codes and the latter one refers to PKI based digital sig-
natures which so far have not gained a strong foothold in Finland.

A pre-requisite for issuing strong identification credentials to an individual is the so-called “first meeting”. This means that the individual has to be come to a physical office to identify himself face to face by using an official identification document such as a valid passport. Similar regulation applies towards the financial institutions for other reasons (Anti Money Laundering, Financing of terrorism). Banks must know their customers (Know Your Customer, KYC) before establishing a business relationship. This is one of the reasons why Finnish banks have naturally adapted the role of an identity provider in the Finnish market for e-commerce services.

Strong electronic identification enables the customer and service provider to make legally binding agreements. To provide Web or mobile based applications that can be used for executing financial transactions or to create new agreements, financial institutions need to identify the customers by using strong electronic authentication.

Other relevant legislation which has an impact regarding native applications is the Data Protection Directive. It defines how sensitive personal information can be gathered, processed and how it must be protected [EUR14]. The Data Protection Directive is extensive and has several implications for almost any type of businesses; in the context this paper the most relevant part is the requirement to securely store any data collected and protect it from abuse.

All European banks are subject to the regulation of European Central Bank (ECB). The main focus of the regulation is in the actual banking business and the goals are to increase competition, harmonize consumer protection and to protect stability of banking systems in the European area. However ECB has also released a recommendation called “Recommendation for the security of internet payments”. This recommendation goes quite far in detail regarding what kind of protection European banking organizations shall have in place for “Internet payments” (referring to any type of internet originating payments, whether Web or Mobile App based). Key points from ECB “Recommendation for the security of internet payments” [ECB13] are the following:

- Protect the initiation of internet payments, as well as access to sensitive payment data, by using strong customer authentication.

- Limit the number of log-in or authentication attempts, define rules for internet payment services session “time out” and set time limits for the validity of au-
thentication.

- Establish a transaction monitoring mechanisms designed to prevent, detect and block fraudulent payment transactions.

- Implement multiple layers of security defense in order to mitigate identified risks.

- Provide assistance and guidance to customers regarding online security best practices, set up alerts and provide tools to help customers to monitor transactions.

Further regulation applies to all applications which handle debit, credit, ATM and Point Of Sales cards. Payment Card Industry Data Security Standard (PCI DSS) governs how card related information can be processed, displayed and how it must be protected in different usage scenarios [PCI13].

As explained in this chapter, in the banking industry it is simply not enough to build secure services and secure mobile applications, these applications must also adhere to extensive regulation and local legislation. A common best practice is to involve appropriate subject matter experts such as an external auditors and the legal department to verify that these obligations have been fulfilled.

### 2.6 Other constraints and requirements

It is important to remember that the new mobile channel is designed in to a specific context. There are a number of existing constraints, requirements and preconditions that influence the design decisions taken later in the thesis.

The constraints in no specific order are the following:

- The constraints for mobile devices include limited processing power (CPU), limited memory, limited battery capacity, limited network connectivity in terms of bandwidth and availability (including switching between networks) and latency, limited features provided by the run time platform, fragmented run time platforms as well as hardware platforms containing different features and capabilities.

- Suggested solution must fit within the existing AS-IS enterprise architecture landscape described in Figure 2.9. The landscape should be extended with mo-
bile related needs and capabilities.

- Needed business services and core systems (including the data) exist, and there shouldn’t be a need to be modified those for adding a new channel. New supporting services might be needed.

- Existing user authentication solutions must be utilized within the applications. It should be possible to support several different authentication mechanisms (such as one time codes, password based, mobile app/sim based). Adding or removing of an authentication solution should not impact or require a redesign for the security solution.

- Internal SOA service layer exists to expose the needed interfaces from the core business systems. Whether the core services reside on legacy mainframe platforms or more modern runtime environments is not significant.

- The bank provides majority of the core business services as a single entity, excluding the need for a federated authentication solution.

- No replication of the data for mobile needs i.e. no secondary copies of the customer registry or any other data. This is to ensure consistency of the data and to adhere to SOA principles and keep ownership of data clear.

- Constraints imposed by the mobile platform providers related to how mobile apps can be developed, distributed and maintained. Further details regarding development, distribution and maintenance of the apps is elaborated in chapter 5.

These constraints make it clear that this work can not be seen as greenfield development but as more of a specific solution for large companies which typically have large number of existing systems and legacy solutions.

Figure 2.6 presents the existing architectural landscape in which the new mobile channel must be fitted in. It also visualizes what is meant by different communication and distribution channels accessing the shared core business functionalities.
Purpose of the channel based approach is to decouple different layers, keep clear ownership and enable the reuse of core business functions across the channels. If a customer wants to perform a payment it can be done by visiting a physical branch office, calling in to Contact Centre, logging in to the Mobile or Web based Netbank or by using the physical Payment ATM machines. In all cases it the payment will be processed and executed by the centralized Payment core systems. If the Bank decides to replace one of the core systems, ideally it will have a minimal impact on the channels since SOA Service layer provides a generic interface to the core systems hiding their implementation specific details.

3 Selecting the appropriate type of Web Services for mobile consumers

Web Service is a general term describing system to system communications over the Web. A common goal for Web Services is to provide interoperable messaging between different software applications running on different platforms [Boo04]. While almost any HTTP based service could be referred as a Web Service, in this context a Web Service refers either to RESTful Web Service or a SOAP based Web Service. Differences and the special characteristics of these two Web Service types are explained in this chapter.
Various attempts have been made for defining and building an interoperable integration protocol to suit the need of distributed systems. First there was Remote Procedure Calls (RPC), then Common Object Request Broker Architecture (CORBA) and after that came Distributed Component Object Model (DCOM), Java Remote Method Invocations (RMI) and finally SOAP based Web Services which use XML for the message exchange. In the modern client-server world HTTP is the natural transport protocol. It is widely supported in different platforms and it is the foundation of the World Wide Web. On top of this foundation has born the concept of RESTful Web Services. RESTful Web Services are not the result of huge corporations setting up big design and standardization committees which is the case for SOAP. REST is more of a conceptual or an architectural pattern which has grown organically based on the fundamental Internet architecture. Roy Fielding, one of the key contributors to the original HTTP specification, initially introduced the concept [Fie00] of REST in his doctoral dissertation.

A case study conducted in 2012 by Vitmar, Vinoski and Pautasso focusing on programmatic Web interfaces reveals the popularity of RESTful Web Services [VVP12]. According to the statistics the researchers acquired from programmableweb.com³ in 2011 75% of the online interfaces were REST based and less than 25% SOAP based. Similar trend can be observed from stackoverflow.com that is one of the largest online discussion and question boards for all types of programming related questions. As of 9.12.2014 there are 28490 REST related questions in stackoverflow.com which corresponds to ~66% of the total SOAP and REST related questions, versus 14755 SOAP related questions (~34%).

There are no definitive figures of how much SOAP or REST is globally used since majority of corporations are not willing to expose any details regarding the numbers of internal or private API:s. However it is rather clear that REST is the current favorite. While the selection of an integration protocol is not a popularity contents, there are implications that should be taken into consideration such as the availability of developers and livelihood of the developer community around the specification, toolsets, and frameworks impacting availability of long term support.

This chapter introduces three different types of Web Services; SOAP based, RESTful with JSON as the content type and RESTful with XML as the content type and analyzes

³ programmableweb.com is a large online repository for listing open interfaces
their strengths and weaknesses for mobile use. The last section of the chapter makes an initial recommendation for selecting the integration protocol based on the information presented so far.

### 3.1 SOAP based Web Services

SOAP is an acronym originally for Simple Object Access Protocol; later on it has commonly also been used for Service Oriented Architecture Protocol [Boo04]. To be specific SOAP is a World Wide Web Consortium (W3C) standard defining how to exchange XML based messages between systems [Mit07]. Another important standard related to SOAP is Web Services Description Language (WSDL) [ChM07]. WSDL is used for describing the interface of the SOAP based services. Both SOAP and WSDL are based on the fundamental XML, XML Schema and XML namespace specifications.

As described, SOAP is based on a stack of standards. It is supported on a wide variety of operating systems and programming languages. While the SOAP specification and other related standards such as Web Services Interoperability Basic Profile (WS-I BP), aim for interoperability, the SOAP critics are saying that it is too complex, requires heavy processing and that the promise of interoperability has not been fulfilled [PZL08].

SOAP based Web Services are agnostic regarding the transport protocol. Messages can be transferred for example over HTTP, SMTP, FTP, MQ and JMS. Fundamentally the request and response messages will contain all required information which is needed for processing them. This enables the services to be stateless. Even if HTTP(s) is the most commonly used transport protocol for SOAP based implementations, the fact that only HTTP POST method is supported means that SOAP is not really adhering to the fundamental Web architecture. The use of a single HTTP method for example breaks protocol level caching as HTTP POST should be used for non-idempotent requests and non-idempotent requests should not be cached. Some also claim that SOAP based services are too slow to develop to satisfy the modern rapid time to market cycle. Time to market is especially in the mobile industry where things change rapidly.

SOAP specification describes a mandatory high level XML message structure for any SOAP based web service. The standard message structure contains a top level element called *Envelope*; inside the envelope are the optional *Header* and mandatory *Body* ele-
ments as illustrated in Figure 3.1. Detailed message content inside the *Header* and *Body elements* and the interface between the service consumer and the service provider is defined in the machine readable interface description, WSDL. WSDL defines detailed information about supported operations that belong to a specific service. It also contains the request and response message structures which are expected and the end point addresses where the messages must be sent [Boo04].

**Figure 3.1 SOAP Envelope**

Purpose of the optional header element is to contain metadata related to the request or response which can be processed by intermediate nodes in the SOAP message exchange before the ultimate SOAP receiver. Metadata in this case refers to data which is not necessarily directly related to the actual payload or business content of the request, but is still necessary to be embedded in to the message. SOAP Header can contain security, routing or service level related information as specified in the several of so called WS-* specifications such as WS-Security, WS-Addressing and WS-ReliableMessaging [PZL08]. These specifications build on top of the SOAP specification similarly as SOAP builds on top of the XML specification.

SOAP Body contains the actual payload of the request or response message and is targeted for the ultimate SOAP receiver. In a simple bookstore service example we could have an operation called “getBookDetails”. The request message for the operation could
be an XML structure containing the ISBN number of a book and the response message would contain an XML structure containing all the relevant details like the name of the book, author, publishing date and the publisher of the requested book.

SOAP Body must be defined in the WSDL but this is not required for the SOAP Header. Continuing with the bookstore example, the WSDL would contain the description of the “getBookDetails” operation with the associated request and response types. The WSDL will also contain the exact data type definitions for the request and response messages. Also names of the individual fields, namespaces, exact data types, ordering of the elements and information regarding whether they are mandatory or optional is included.

WSDL supports embedded XML schema or references to external XML schema files for defining the message format. Referencing to external schemas is a useful feature if the commonly used data types of an organization have been previously been described. The use of external schemas enable re-use for the data types between services that utilize the same fundamental data types. Use of XML schema enables SOAP messages to be schema validated to verify that they are correctly formatted and can successfully interpreted by the sender and receiver [PMP10]. Schema validation eases the development efforts as it simple to pinpoint if it was the sending or receiving party who didn’t format the message according to the previously made agreement described in the WSDL.

Disadvantages of SOAP based services include verbose messages that increase the size of the message exchange. Parsing of the XML messages in the mobile devices is fairly slow and resource consuming [GiT11]. The SOAP specification in general is rather complex and requires in depth knowledge from the developers. Complexity also sets requirements for tooling, it is a huge effort to use SOAP with the platform primitives (HTTP client and XML parser) if designated libraries are not available.

### 3.2 RESTful Web Services

REST which is short for Representational State Transfer is a term introduced by Roy Fielding in his famous doctoral dissertation from the year 2000 [Fie00]. REST is not a standard or a specification but a set of architectural principles which enables building of scalable distributed applications [PZL08]. RESTful services are focused on addressing the resources of a system via Uniform Resource Identifiers (URI). Resource states can be managed over HTTP by client applications written in many different programming
languages as long as they contain support for the basic HTTP standard.

Typically a resource representation will reflect the current state of a resource and its attributes at the time when a client application requests it [RiR07]. Resource representation in this sense is only a snapshot at specific moment time. This resource representation could be for example a record in a database, an image or document in a file system or a detailed part of a domain specific data model.

The structure of a RESTful request/response message is divided into three independent sections as described in the HTTP specification; [Fie99] 1) HTTP headers contain operation related parameters and additional metadata 2) URI which identifies the resource to be operated and 3) optional payload which contains the actual data related to the request or response.

RESTful services were introduced only two years later than SOAP based Web Services. The concept is not new but REST has really gained momentum in the last couple of years [VVP12]. One likely reason is that more and more developers have become frustrated with the complexity of SOAP and the general mismatch between SOAP and HTTP protocol. Big online vendors like Google, Facebook, Amazon and Twitter are primarily focusing on RESTful services for providing their online API:s.

RESTful Web Services are tightly bound to the use of HTTP as the communication protocol. HTTP verbs GET, POST, PUT and DELETE are most commonly used to indicate what kind of operation is to be performed for the selected resource [Fie99]. The resource to be operated is identified by the URI [RiR07]. Additional HTTP methods (CONNECT, HEAD, OPTIONS, PATCH, TRACE) exist but they have not gained a wide usage within the RESTful context.

Basic usage of the HTTP verbs is the following [Fie99]:

- To retrieve a resource, use **GET**.
- To create a resource on the server or to initiate server side data-processing, use **POST**.
- To change the state of a resource or to update it, use **PUT**.
- To remove or delete a resource, use **DELETE**.

Nature of the different HTTP methods means that for some operations there is no re-
quest payload, since all the relevant information is included the HTTP method and request URI. Using the previously mentioned bookstore service as a RESTful version, requesting the information related to a book with an ISBN number ISBN123456 from the bookstore service could be done by issuing the following HTTP request: 


Comparing this request to a corresponding SOAP request reveals the simplicity and minimal request size of RESTful read only operations. The SOAP based request would also have a specific URI for the operation and specific XML based request message structure embedded in a SOAP envelope.

RESTful Web Services are agnostic to the message format used. Typically the supported content types are predefined and the client can attach a specific Accept HTTP header to the request to indicate the desired content type. Two popular messaging formats used with RESTful Web Services are XML and JavaScript Object Notation (JSON). Usage of XML and JSON based messages and their specific characteristics are explained in more detail in the following sub-sections.

RESTful services typically rely heavily on standard HTTP status codes. HTTP 200 is used for OK response; HTTP 500 indicates an internal error inside the service while HTTP 400 indicates that request sent by the client was malformed. HTTP status codes are also used for example for redirecting, indicating caching related information and for various other purposes [Fie99]. Benefit of the standard status codes is that common internet based infrastructure can interpret these codes and act accordingly when needed [Fie00].

Resource oriented nature of REST also causes some interesting questions which are not widely discussed in the relevant literature. For example can everything be modelled as resources? Concrete data like the bookstore related examples above are easy to identify as resources. But modelling actions in to resources is already a bit more challenging. For example; how to model an authentication, payment or a chain of actions i.e. a long running business process as a resource?

Experience from working with RESTful services within a complex IT environment shows that it is not always feasible to expose the concrete data structures as resources. A popular approach is to increase the level of abstraction and provide a high level RESTful facades, which can for example hide the transactional needs (or associated business
logic) related to the resources.

Using a payment between two accounts as an example; it is not feasible to open up direct access to account ledgers as RESTful resources and trust the service consumer to guarantee transactionality. The consumer might fail to make the required bookings on both accounts when transferring money from account to another. Better approach would be to build some kind of top level transaction which will always provide expected outcome. This top level transaction which is responsible for performing several activities can hardly be described as a resource. Certain level of abstraction is needed in order to model the real world in to RESTful resources that are convenient to use.

Disadvantages of using RESTful services as the integration pattern commonly relate to the abstract level of the architectural pattern. There are several interpretations of what it takes for a service to be truly RESTful. REST is also lacking the mature tool set of based on standards which have developed around SOAP. There is no formal interface description language, message level security specifications, transaction support nor reliable messaging.

### 3.2.1 XML based RESTful Web Services

SOAP based Web Services and the general interest towards XML in the early 2000’s have been the major influencers for starting to use XML as a message structure with RESTful services [RiR07]. XML based RESTful services are still used and many RESTful services offer both JSON and XML based content types but the general trend is leaning more towards JSON.

Benefit of using XML as the messaging protocol are the mature standards around XML. XML namespaces can be for example used to specify the context of an “Amount” field. In some context it could refer to the amount received and in another context amount paid. Also the XML related security standards can be used to protect the message content in the cases when an actual message payload exists in addition to the HTTP method and URI.

Disadvantages XML based messaging include increased message size and increased CPU consumption for parsing of the messages both in client and server [GiT11].
3.2.2 JSON based RESTful Web Services

As defined in json.org, “JSON (JavaScript Object Notation) is a lightweight data-interchange format”. JSON is based on a subset of the JavaScript (ECMA-262) standard. It is programming language agnostic and utilizes name-value pairs and lists of name-value pairs as the primary data structures. These universal data structures can be found from almost all programming languages making it simple to utilize JSON in heterogeneous programming language environments.

Typically many modern Web applications have the requirement of integrating the user interface in to the back end services to show dynamic content. This has been one of the driving forces of using JSON based messages over RESTful services. JSON is the natural way of passing data to JavaScript functions that are commonly used to render asynchronous content to modern Web 2.0 Web applications. JSON as such in not bound to RESTful services, but it is a general purpose data exchange format.

Current trend is favoring JSON based RESTful services since they are seen as a simple and lightweight option for XML based messaging. Indeed the JSON syntax provides smaller message size [HFP11] and is faster to parse by the sender and receiver and consumes less battery from the mobile device as proven by measurements done by Bruno Gil and Paulo Trezentos [GiT11]. The trade-off for the convenience and good performance is in the lack of mature JSON related standards that exist in the XML world [Ser12]. Decreased message size along with decreased CPU and memory usage are factors that clearly make JSON the optimal candidate as the message syntax to be used in a mobile environment.

3.3 Comparing the features of SOAP based and RESTful Web Services

Different features and properties of the presented integration protocols are gathered in to Table 3.3 to enable comparison feature by feature and to ease the protocol selection process. The table can be used to analyze the tradeoffs associated with the selection of the integration protocol.

So far the security features of the integration protocols or the capabilities of the selected mobile platforms have not been presented yet. They are still both important factors in selecting of the optimal integration protocol so they are included in the table. Security features of the integration protocols are presented in Chapter 4 and support for the
SOAP based and RESTful services in the modern mobile device platforms is analyzed in more detail in Chapter 5.

<table>
<thead>
<tr>
<th>Capability</th>
<th>SOAP based Web Services</th>
<th>RESTful Web Services</th>
<th>Additional comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple transport protocols</td>
<td>x</td>
<td>-</td>
<td>Not extremely relevant if the services are targeted only for mobile usage since HTTP is the de-facto transport protocol.</td>
</tr>
<tr>
<td>Standard transport layer security</td>
<td>x</td>
<td>x</td>
<td>SSL when using HTTP, additional mechanisms available when using SOAP with other transport protocols.</td>
</tr>
<tr>
<td>Multiple messaging formats</td>
<td>-</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Standard interface description language available</td>
<td>x</td>
<td>-[*]</td>
<td>[*] WADL and some tool specific description languages exist in the RESTful context, but none of them have gained wide acceptance.</td>
</tr>
<tr>
<td>Standardized message level security</td>
<td>x</td>
<td>-</td>
<td>Elaborated in more detail in chapter 4.</td>
</tr>
<tr>
<td>Standardized support for transactions</td>
<td>x</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Support for reliable messaging</td>
<td>x</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>
Table 3.3 Comparison of capabilities and features between SOAP and RESTful Web Services

<table>
<thead>
<tr>
<th>Capability</th>
<th>SOAP based Web Services</th>
<th>RESTful Web Services</th>
<th>Additional comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest message size (less bandwidth used)</td>
<td>-</td>
<td>x</td>
<td>HTTP compression can decrease the gap, but still RESTful messages will have less overhead</td>
</tr>
<tr>
<td>Less CPU consuming parsing/processing (better battery life in mobile)</td>
<td>-</td>
<td>x</td>
<td>Elaborated in more detail in chapter 5.</td>
</tr>
<tr>
<td>Native support in mobile platform API:s</td>
<td>-</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Active developer community in the mobile context</td>
<td>-</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.3 gathers the properties, features and capabilities of SOAP and REST based Web Services and provides some information about the possible trade-offs of each pattern. For example if message level security is a must requirement, then SOAP is the only pattern supporting that. The tradeoff is that there is no native support in the mobile platform API:s for SOAP. Or vice versa, if multiple messaging formats is the key requirement, then the tradeoff is that only one transport protocol is available, as SOAP can not be used.

Pautasso, Zimmerman et al have done a similar comparison without the mobile aspect. Their conclusion is that SOAP based services are better suited for professional enterprise application integration scenarios while REST is more suited for tactical ad hoc integration over the Web [PZL08]. When the results are combined with the mobile aspects such as message size, native support in mobile platforms and battery consumption, RESTful services start to seem even more appealing for integrations over the Web, let it be for tactical or strategic purposes.
Selecting the appropriate integration pattern should be based on the actual requirements of the system that is being built. The relevancy and importance of the capabilities and features listed will most likely differ per organization or possibly even between different systems within the same organization. If Table 3.3 is to be used as a reference for decision making, the decision makers should weigh the different capabilities according to their specific needs and requirements. Security related aspects and support in the selected mobile platforms will be described in more detail in the following Chapter 4.

3.4 **Recommended pattern for Mobile use**

Taking in consideration the mobile related constraints presented in chapter 2.6 and the descriptions of SOAP and RESTful services in this chapter, the conclusion is that RESTful Web Services with JSON are best suited for the mobile usage. Smaller message size, decreased battery usage and the fact that JSON is recognized as a native data type requiring less heavy processing in the mobile platforms are technical facts in favor of RESTful Web Services.

RESTful Web Services have achieved sort of a de-facto status for internet based solutions. Even if one could say SOAP is much more mature and comprehensive specification when comparing them feature/capability wise. This de-facto status means that the REST is the skillset that is going to be available in the developer market. Looking in to the future RESTful Web Services is the technology which is going to have the more active development community. Future incremental development steps in the integration protocols are most likely going to be based on the REST ideology until someone comes up with a revolutionary idea that offers a completely new alternative.

This recommendation will be updated in the coming chapters regarding security features and features found from the mobile operating systems.

4 **Security features of SOAP based and RESTful Web Services**

RESTful and SOAP based Web Services differ significantly in terms of the security standards and features available. SOAP has several mature standardized security extensions while RESTful Web Services mostly rely on best practices and implementation and vendor specific solutions.
WS-Security is the de-facto security framework to be used with SOAP [Law06]. WS-Security builds on top of XML digital signature (XMLDSIG) and XML encryption (XMLENC) standards that are the fundamental building blocks of XML based digital signatures and encryption [Law06]. Several other even more evolved security specifications have been developed on top of WS-Security. These include WS-Trust, WS-SecureConversation and WS-SecurityPolicy as described in Figure 4. These extensions are targeted for specific use cases which are not relevant in the context of this thesis.

<table>
<thead>
<tr>
<th>WS-SecurityPolicy</th>
</tr>
</thead>
<tbody>
<tr>
<td>WS-SecureConversation</td>
</tr>
<tr>
<td>WS-Trust</td>
</tr>
<tr>
<td>WS-Security</td>
</tr>
<tr>
<td>SOAP</td>
</tr>
<tr>
<td>XML Signature</td>
</tr>
<tr>
<td>XML Encryption</td>
</tr>
<tr>
<td>XML</td>
</tr>
</tbody>
</table>

*Figure 4 describes the relation between XML and SOAP related standards.*

RESTful Web Services with JSON are security-wise in a completely different situation compared to SOAP. As JSON and XML differ feature-wise it is impossible directly utilize any of the existing XML security features with JSON. XML contains attributes, object references and data types which are used in WS-Security. The basic principles of public key cryptography that are behind the XML security standards could be utilized for JSON based services. That would however require significant standardization efforts. Additional challenge comes from the nature of REST as explained in the previous chapter. The request or response related data might be divided in to several parts of the HTTP message including headers, URI and the JSON payload. A protection only covering the JSON body would not be sufficient.

One security solution that is available for both Web Service types is Transport Layer Security (TLS) also commonly referred as SSL or HTTPS [Ser12]. Basic TLS provides encryption of the message exchange and identification of the service provider. Optionally authentication of the service requestor can be achieved in the so called mutual au-
The authentication scenario, where the client also holds a dedicated certificate [DiR08]. One could argue that SSL should be a sufficient solution for Web Services based systems, since currently SSL is still one of the security cornerstones of most of the Web based solutions. While this claim is true for SSL being the widely used by current Web applications, it is not a very future-proof approach. When building new architecture for a system which is expected to have a long lifecycle, it should at least include the capability of adding new security features later on as needed, if not included right away from the beginning. Most important areas for improvements would be message level encryption and message level signatures, which can protect the message exchange even if the protection provided by transport protocol is compromised.

Having SSL as the only primary mean for protecting the communications is an increased operational risk, and an approach that should be avoided with business critical systems. One critical bug in the SSL libraries can completely destroy the security of the system and require services to be completely halted, as was seen with the Heartbleed vulnerability in April 2014 [CVE14].

Rest of this chapter focuses on introducing and comparing the details of available security solutions for SOAP and RESTful Web Services. In the end of the chapter there is a recommendation for the security solution building on top of the recommendation from Chapter 3.

4.1 Security of SOAP based Web Services

SOAP based Web Services can be secured either via transport layer security (SSL) or WS-Security based standards. Fundamental difference of these mechanisms is that WS-Security provides end-to-end message level security while transport level security such as SSL or a VPN only offers point-to-point security. Point-to-point security means that the message is protected only between the two immediate nodes which communicate directly with each other, possibly leaving the messages vulnerable in the intermediate nodes as displayed in Figure 4.1.
End-to-end message level security means that the request and response messages contain all the security related information and protection. It also means that the security of the solution is completely independent of the network topology or transport protocol. This makes WS-Security a very desirable model. It guarantees that as long as the sending and receiving systems have been built according to WS-Security specification, messages can be routed through any number of intermediate nodes in any kind of network topology. Trade-offs for this flexible model include significantly higher performance requirement for systems processing the messages, increased message size and a fairly complex implementation model.

WS-Security specification describes how message integrity and confidentiality can be achieved by using digital signatures or encryption [Law06]. Original sender of the message can be identified via an embedded security token such as a X509 certificate, SAML assertion or Kerberos tickets [NNA09]. WS-Security builds on top of XML Digital signatures (XMDSIG) and XML encryption (XMLENC) standards [Law06] with small adaptations to fit the SOAP envelope structure. WS-Security specification is rather flexible regarding how the protection mechanisms can be used; encryption and signature can be used independently or together and it is possible to select only specific parts or the entire message to be protected [Law06].

Figure 4.1 Point-to-Point and message level security. SSL or other point to point transport layer security is broken in the intermediary node which will have the possibility to read or modify the message. Message level WS-Security guarantees end-to-end security in the entire processing chain. Green envelope presents message being safe and red envelope indicating message being vulnerable.
WS-Security relies heavily on Public Key Cryptography. The fundamental capability offered by asymmetric public/private key pairs is the ability to encrypt or sign data with one of the keys and the decrypt or verify signature with another key [Vac04]. Neither of the keys can be used alone to perform both actions. To be able to use Public Key Cryptography in an organized manner certain amount of infrastructure including people, processes, policies and systems are needed to manage the keys and their usage [DaM09]. This is referred as Public Key Infrastructure (PKI). PKI introduces the concepts of a Certificate Authority (CA) and Registration Authority (RA). RA is responsible for verifying the identity of the entity that possesses the public key and the CA is responsible for binding this entity to a digital certificate containing the public key [DaM09]. While the PKI based systems are currently regarded as being very secure, the required infrastructure is fairly complex and expensive.

Figure 4.1 displays the nested element structure of a SOAP message which is digitally signed according to WS-Security specification. When comparing the signed SOAP message in Figure 4.1 to the empty SOAP envelope presented in Chapter 3.1 (Figure 3.1) it is obvious how much additional complexity the digital signature scheme brings. One could also argue that the picture is a rather good argument for the SOAP critics who are worried about the additional overhead of the SOAP protocol and general complexity of the WS-Security.

Regardless of the complexity of the digital signatures, they have some distinct advantages when done according to the specification. Sending and receiving systems are not required to know anything about each other as the message is completely self-describing regarding the keys and algorithms used. When message sender signs a SOAP message with his/her private key and attaches the public key as a digital certificate to the message, the message receiver can verify the signature and be sure that the signature has been created with the private key matching to the public key in the certificate. If the message receiver trusts the CA which has issued the digital certificate, he can be sure that the content of the message has not changed during transit and that the message has originated from the holder of the private key. Furthermore the receiver can store the received request to be used as a proof that the message has originated from a specific sender holding a specific private key.
When the received message is stored persistently or logged as-is it can be later on used as a proof of that the message must have originated from a sender possessing a specific private key. This is what is generally referred as non-repudiation [PKP08]. The sender of the message can not deny sending of the message since it has been signed with his/her personal private key. This can be proven any time by recalculating the digital signature found from the message by using the public key of the sender which is included in the original stored message.

Since digital signatures created with asymmetric keys that are issued by an approved identity provider are considered as a form or strong electronic authentication it is very important to store the private keys in a secure way. Stealing a public-private key pair...
means stealing of an identity. Once noticed, this threat can be mitigated through the tools offered by PKI such as Certificate Revocation Lists (CRL). CRL in essence is a blacklist for compromised certificates not to be accepted even if the certificate data is valid. Due to the sensitive nature of the private keys it is often considered to be unsafe to store them in user devices which are directly connected to internet and vulnerable to different kinds of attacks. To limit the risks digital certificates are stored in special hardware security tokens which only allow signature generation in the hardware token, but never allow the private key to be exported outside of the device. Some of the modern mobile devices contain an isolated hardware security module called Secure Element which can be used for storing such keys. Secure Elements are elaborated in more detail in Chapter 5.

4.2 Security of XML based RESTful Web Services

XML based RESTful services can partially benefit from the mature XML related security standards mentioned in the previous chapter. The most important standards are XMLDSIG, which can be used non-repudiation and message integrity, and XMLENC which can be used to assure confidentiality. By using X509 certificates to perform these operations the user can prove his identity by attaching the X509 certificate to the signed XML request. These mechanisms are valid in the cases where the request and response message contain a XML payload that can be protected by signing or encrypting. Problems arise with HTTP GET and DELETE methods which do not contain a payload [Fie99]. If there is no message to be signed, because the request is for example a HTTP GET to fetch a resource from a specific URL, it is not possible to sign or encrypt anything and attach an X509 certificate. This in turn leads to a situation where the service provider can not identify the service requestor.

Same problem exists for verifying the identity of the service in case of empty response messages. Most typical empty response would be for example HTTP response code 302 which is indicating a redirect [Fie99]. There would be no way to authenticate the service provider’s identity that has provided this response code since the response does not include a payload that could be signed.

Additional problems arise due to the structure of the RESTful messages. As explained
in Section 3.2, a RESTful request/response consists of three sections. XML can be only used in the payload, meaning that the two others (HTTP headers, URL) would be left unprotected. It would be possible to include header and URL related information to the payload, but that would increase message size, bring additional complexity and strongly violate the REST ideology.

Conclusion is that the benefit provided by using existing XML related security standards with XML based RESTful Web Services is very limited. The main use cases are HTTP POST and HTTP PUT based services which contain a request/response payload. Using different security solutions for different request types would create significant additional complexity. The recommendation is to protect XML based RESTful services with similar means as explained in the next section for JSON based restful services.

4.3 Security of JSON based RESTful Web Services

RESTful Web Services do not have a built in security model. Although Roy Fielding mentions “enforcement of security” several times in his dissertation there are no concrete requirements or instructions how to build secure RESTful services [Fie00]. Serme et. al. conclude that RESTful Web Services lack the means to describe security related meta data requirements and that most of the public RESTful Web Service implementations rely on ad hoc and custom security solutions [Ser12].

As mentioned in chapter 2.1 one requirement for RESTful Web Services is to be stateless. This poses new requirements for the security solutions that can be used. Traditional model used by many Web applications where cookies are sent by the customer’s browser for identifying an authenticated session held by the Web Server must be replaced with something else. Since the requirement is to be stateless, one common approach for replacing server side sessions is a token based approach [BH14]. The token must self-describing, have a limited lifetime, be cryptographically protected and contain all information necessary for the validation of the token. Previous trust relation must be established between the party who issues the token and the party who validates the token if they are not the one and the same. Since the token is cryptographically protected, “previous trust” generally refers to a key exchange where the validating party must have access to the crypto keys which are needed in the token validation. This can be done via a shared key store or by physically exchanging the keys.
Challenges of the token based approach relate to managing the lifecycle of a token. Once the token is issued it will have a fixed validity period which can not be altered. This generates the need for some supporting capabilities to enable similar session management that is seen with traditional Web applications today. For example how to continue an active session if the token is about to expire, or how to deactivate a token if the customer decides to end the session.

Two security related standards which are often brought up in the RESTful context are OpenID and OAuth. Both of these also utilize a token based approach. OpenID is a standard for federated authentication [Fer07]. It has two main parties, identity provider and relying party. This means that the user registers with the identity provider and after that he can authenticate to any relying party who has established a trust relation with the identity provider.

OpenID is not seen as an attractive option for the Bank. European Anti-Money Laundering regulation contains strict Know Your Customer (KYC) requirements. KYC requires that the institution must meet the new customer face to face before strong electronic identification credentials can be issued. So the Bank already acts as an identity provider and there is no further need for federation within the services that the Bank provides. Furthermore current OpenID versions are strongly tied to a HTML based web user interface for selecting the identity provider [Fer07]. Although technically this could be considered to still be RESTfull, since its http/https based, a fixed user interface fits poorly in to an integration scenario where the desire is to use services via API:s and have full control over the UI.

OAuth is a federated authorization protocol [Har12]. As the official description explains, it is targeted for authorization which is different from authentication as previously explained. Despite of OAuth being an authorization protocol it still contains some elements that can be used for authentication purposes. Typically OAuth is used for delegating authorizations to a third party application [Ham10]. For example a third party RSS feed reader might use OAuth to request authorization from the user to read a secured RSS feed on behalf of the user. Although the main focus of OAuth is the specific use case where a resource owner authorizes a third party to access a resource from a provider, there are other supported scenarios. One of them is the so called two-legged OAuth that enables authentication between a requester and a resource provider [Ham10] which corresponds with the authentication scenario presented in this thesis.
OAuth 1.0 is currently the only accepted IETF standard for RESTful services which also contains message level protection with digital signatures or Hashed Message Authentication Codes. Unfortunately there is a specific issue with the content that can be signed, the HMAC or RSA signature does not cover payload data for HTTP POST or PUT methods with typical RESTful content-types such as XML or JSON. Only “application/x-www-form-urlencoded” content type is supported [Ham10]. This basically limits the content to the typical key value pairs produced by HTML forms. Latest version OAuth 2.0 has completely dropped support for message level protection. This has led to some controversy and as a result OAuth 2.0 specification leader Eran Hammer-Lahav has resigned from the workgroup and publicly criticized weak security of the specification.

Although OAuth is heavily used by the Internet giants such as Google and Facebook, several security concerns have been presented. Feng Yang and Maharam have done a root-cause analyze for the OAuth security weaknesses and conclude that there are several threats in different steps of the protocol [FeM13]. OAuth is not considered secure enough for financial transactions by the Banks IT Security department either and therefore it will not be analyzed deeper in this paper.

One typical approach to RESTful authentication is a HMAC based custom authentication scheme with shared keys, utilized by Amazon S3 and Microsoft Azure services for example. This scheme is based on Hashed Message Authentication Codes which are created by calculating a keyed-hash sum from a message and a message key\(^4\) [Kra97]. The use of individual keys per user is the essential element that enables HMAC signatures to be used for authentication. In the HMAC authentication process the service consumer must first obtain a secret key that is shared with the service. Issuing the keys happens in a registration or an enrolment process. The issued secret key is later used for calculating the Hashed Message Authentication Code, initially by the sender to protect the message and prove his identity and later by receiver to verify the integrity of the message and the identity of the sender.

One advantage of HMACs is that Secure Hash Algorithms (SHA) are much faster and less CPU consuming than Advanced Encryption Standard (AES) and Data Encryption

\(^4\) Message key is often also referred as API key or secret key in the RESTful context
Standard (DES) based functions [SiS11]. AES and DES based functions are typically used in asymmetric cryptography. This is especially relevant in the mobile context where the resources of the application execution environment are limited.

The process of generating a HMAC based signature for a RESTful request consists of a custom canonicalization protocol which is specified by the service provider. The canonicalization protocol must define how HTTP headers, request URL and request payload are normalized. In practice it defines how different parts ordered, encoded and concatenated to generate one long string that represents the entire request (also referred as canonical request). After that the HMAC is calculated by the client from the canonical request by using the shared secret key. The HMAC digest is then attached as a HTTP header with an identifier referencing the specific key that was used for calculating the digest so that the receiver knows which key must be used for the verification process.

The HMAC needs to be generated for every single request, so this fulfills the stateless requirement for RESTful Web Services. When the service provider receives the request, the first action is to fetch the secret key based on the identifier provided by the customer. After that the same canonicalization and HMAC calculation processes are applied as previously done by the sender. The final step is to compare the HMAC digest provided by the customer to the digest calculated by the service provider. If the two digests are identical, it means that the request must have originated from the holder of the key referenced. The integrity of the request has been verified and the sender of the request is authenticated.

Internet Engineering Task Force (IETF) has established a working group called “JavaScript Object Signing and Encryption”. This working group is working to develop similar security standards for JSON that exist for XML/SOAP. One of the most crucial things that are needed to get interoperable digital signatures across multiple platforms is a mature canonicalization mechanism that would also addresses the requirements of HTTP protocol.

Purely JSON related security standards are not going to solve the problem for RESTful services. The fundamental complexity comes from the fact that the security solution has to address both a transport protocol and a message format due to the tight coupling of RESTful Web Services in to HTTP protocol. This is a likely reason why mature RESTful security specifications still don’t exist to this day.
4.4 Conclusion and recommended security solution

Based on the information presented in this chapter it is strongly recommended to build message level protection in to the solution in addition to the standard SSL transport protection. The main alternatives are digital signatures according to the WS-Security specification for SOAP and Hashed Message Authentication Codes for RESTful services. RESTful services with XML can offer protection only for limited use cases. WS-Security has an advantage being a proven standard. Hashed Message Authentication Code is a well-established standard as well, but applying it to the different parts of a HTTP request is a custom practice demonstrated by Amazon and others.

Solutions based on asymmetric cryptography are generally regarded as being more secure than the symmetric counterparts, due to the nature of the keys. With asymmetric keys the private key is never shared which is one of the enablers for non-repudiation. As long as the holder of the key stores it carefully, there can be no doubt of who has used the key. With symmetric keys the secret key must be shared between the customer and the service provider. This means that key has to be initially securely transported between the two, and in case of dispute it might be difficult to prove which party has actually used or leaked the key.

Due to the nature of the asymmetric and symmetric keys, SOAP based solutions with WS-security can offer slightly better security than RESTful Web Services with HMACs. The difference is mostly related to the non-repudiation related aspects since both mechanisms provide message integrity and authentication.

Both asymmetric and symmetric cryptographic security solutions share the same characteristics regarding configurability; it is usually rather trivial to improve the security level of the system by increasing the key length and selecting a stronger variant of the algorithm in use.

Most likely vulnerabilities for digital signature or HMAC based solutions are not related to the algorithms being brute forced or broken. The weakest link in the process is typically related to the key handling process. If the attacker is able to penetrate the initial key exchange process, or steal the persistently stored keys from mobile devices or service provider, both solutions are similarly vulnerable.
4.5 Web Services related security considerations

Security of the Web Services exposed in to open internet can not be guaranteed only by selecting and applying the right security standard to the solution. There are several implementation related and operational considerations which should be taken in to consideration. The Web Service endpoint acts as a gateway to the internal core systems which store one of the most valuable assets that a company possesses: the data. It is crucial to ensure that this gateway can only be used for accessing certain limited data according to rules and policies specified in the authentication and authorization requirements. This section lists certain best practices and solutions to make sure that the published Web Services are properly secured. These practices are gathered from OWASP Top 10 lists, European Central Bank recommendation for secure mobile payments and from recommendations by Information Security companies.

Server Hardening is recommended to limit the attack surface available in the environment that hosts the Web Services [ECB13]. Typical means for hardening include: use the servers for only one purpose (hosting the Web Service), block access to all other TCP ports than the one the Web Service is listening to, disable unused operating system services, install latest security patches, limit admin access only from predefined locations and make the initial installation of operating system and patches in a safe network environment. Any clues about the used hardware or operating systems or software libraries (such as informational HTTP headers) should be hidden to make it more difficult to reuse known weaknesses and vulnerabilities of the components.

Injections can happen in multiple ways. The most famous is the SQL injection where the attacker is able to inject additional parameters to the SQL query executed by the database [OWA13]. Other scenarios include for example JavaScript injection in to the JSON payload to be executed by the server or XML content inside [PMP10] the JSON request which might be later on executed in the SOAP based SOA layer\(^5\). Any data sent by the service consumer which is stored and potentially later on viewed with a browser contains the risk of injections. Malicious JavaScript embedded in to the data might be evaluated and executed by the browser of the person who is viewing the data. Typical protection means include input data validation and filtering, sanitizing all user input,  

\(^5\) Billion laughs is a typical example of an XML based attack, a simple XML document contains cyclical references and when the parsers loads the document it expands to consume all available memory of the system.
escaping SQL, HTML, JavaScript, XML and JSON related special characters and configuring memory usage limitations for parsers.

Authentication and session management is a sensitive area that needs to work seamlessly end-to-end [OWA13]. It’s not sufficient to have a strong authentication protocol if the session management of the system leaks. Stealing of sessions must be made difficult, particular challenge in the mobile environment is that the IP address of the customer can change any time due to the nature of the mobile networks or switching between Wi-Fi and a data connection. Typical means include a strong session protocol which is seamlessly integrated to the initial authentication, use of SSL, active logout which invalidates the session, destruction of single-sign-on tickets and message level security with digital signatures or message authentication codes. When digital signatures or HMACs are used the attacker needs to have a local access to the targeted device to steal the key used for signature, while with ordinary session identifiers it might be sufficient just to capture network traffic [ECB13].

Insecure Direct Object References is a typical problem for RESTful services [OWA13]. As shown in the bookstore example earlier, the address to fetch a book details contained directly the identifier of the book. Now imagine the same example for fetching the details of a bank account: “HTTP GET www.mybank.com/account/FI123456789”. It shouldn’t be possible to fetch any unauthorized data by guessing identifiers. Primary mean for preventing direct object references are properly implemented authorization rules where every request is authorized against a listing of the assets available for the customer, and indirect object references⁶. Indirect object references that are artificial identifiers which only might have a lifetime during one session can also be used as a mitigation measure [OWA13].

Authorization and business rules should be implemented and enforced server side and all input coming from the mobile applications should be treated as untrusted. There are no definitive ways to identify the calling application since the application and the Web Service endpoint have been made publicly available. One could reverse engineer the mobile application, copy the decompiled source code, and make a version that circumvents business rules built in to the application.

---

⁶ Indirect references violate against the RESTful principles since they can not be cached or directly linked as a bookmark [Fie00]. However the nature of the content provided by the financial services is sensitive and no automatic caching should be allowed anyway and bookmarking is not relevant in the context of native mobile applications.
Sensitive data exposure can happen in surprising areas even if the authentication and security solutions are properly build [OWA13]. All HTTP response messages should have appropriate HTTP headers indicating that no caching is allowed so that the data doesn’t end up stored in network devices. Previously mentioned indirect object references also help here. Web Servers typically gather access logs of all incoming requests, if the request URL contains sensitive information it might end up visible for unauthorized personnel. The strength of the used encryption algorithms should be appropriate. Encryption keys should be adequately protected and there should be a rehearsed process in place to change and replace the keys if they have leaked. Risks and mitigation measures regarding data exposure in the mobile devices and applications are explained in more detail in Chapter 5. SSL should be always used to maintain confidentiality.

Distributed Denial of Service (DDoS) attacks aim to make a service or a network unavailable by sending more traffic to the system than it has resources (network or computing) to process [Mas13]. Typically they are implemented by botnets with hundreds to hundred thousand nodes. It is impossible to prevent DDoS attacks but the impact can be mitigated. Most importantly the internet facing systems must be isolated so that in case of a DDoS attack other business systems can continue to operate normally. A typical mean for isolating is by configuring throttling limitations to the Web Services layer. For example the Web Service interface might process maximum 200 requests per second and throttle or discard anything above that. This way it’s possible to safeguard the business critical core systems and agree on a maximum level of traffic they will receive. Mitigating the impact of a DDoS attack against the mobile channel will require collaboration with the internet operator(s) to stop the traffic early enough. If the attack is mitigated only when the traffic reaches the bank it is usually too late. It is possible to send so much traffic that even if the throttling mechanism would be able to handle the load, communication lines will choke and all services behind these communication lines become unavailable.

\footnote{A severe example of not isolating the systems properly was seen in Finland in 2014. A DDoS attack against the internet portal of OP Bank brought down cash withdrawals from ATM’s and card payments which used online cover reservations [Tiv15]}
5 Remote communication and security feature support in mobile platforms

This chapter investigates the selected mobile platforms Apple iOS, Google Android and Microsoft Windows Phone. The aim is to discover what kind of support exists for remote communications and what type of local security mechanism are offered by the platforms to be used as a part of securing the remote communications. The goal is to find a common denominator for all platforms to be able to offer a single solution which can be utilized by applications running in any of the platforms.

The main focus of the comparison on the protocol level is SOAP with WS-Security support and RESTful / JSON support. Security-wise the focus is in supported HMAC algorithms, handling of secret keys via platform key stores, advanced features such as secure elements, capability to manipulate SSL handshake to inject extra security measures and other additional security features which might prove out to be useful.

First some background knowledge regarding the mobile platforms is provided. All of the mentioned platforms come with a number of security features intended for securing the local execution of the native application such as sandboxing, signed binaries, protection of local data and explicit grant of privileges for the application by the user (either at installation or run time)[App14, Goo15, Mic15a]. There is slight variation in the implementations between platforms but the fundamental principles are rather closely aligned.

Another significant contributor to the overall security of the system is the secure distribution channel for the applications. All of the platforms have dedicated App stores for publishing and distributing applications developed by third parties [App15a, Goo15a, Mic14]. There are multiple slightly varying processes to go through before an application can be published. The publisher must enrol to the app store to be authenticated, the app to be published is subject to a review and there are multiple guidelines which define requirements by the platform owners that the mobile applications must adhere to. These processes are in place to limit phishing, spreading of viruses and fake applications and to build consumer trust towards the official market places.

Regardless of the efforts by the platform vendors, mobile malware exists and is an increasing risk as reported by F-Secure and other security analysts [FSE14]. There are
Root / Jailbreak kits available for both iOS and Android platforms. These kits essentially allow the user to remove all platform security limitations and obtain full access with root privileges to the device. Obviously the users who disable the platforms built in protection mechanisms expose themselves to a much greater risk.

Comparing the detailed security features of the app stores and local mobile execution environments between the different ecosystems would be an interesting topic, but it is not in the scope of this thesis. Last section of this chapter lists common best practices which can be used to enhance the security of the mobile application and some not so well known pitfalls that should be avoided.

5.1 Apple iOS

Apple iOS is a mobile operating system developed by Apple Inc. It shares the same Unix-like Darwin foundation which is the core of Apple’s desktop computer operating system OS X [LHC11]. The user interface in iOS is optimized for touch screen devices [Der14]. iOS was introduced in 2007 and it is currently supported by all of the apple touch screen device families such as iPhone, iPad and iPod touch. Xcode is the only development environment for iOS SDK and Objective-C is the primary programming language for iOS [Der14].

Remote communications support in iOS is heavily focused towards HTTP based RESTful communications. iOS online API library confirms that the platforms does not provide any built in SOAP / WS-Security support [App15d]. There are some open source SOAP tools available, but most of them are not actively maintained and have received poor ratings on the developer discussion forums. Same was confirmed within the Bank in a proof-of-concept where some of the libraries were evaluated. Multiple bugs were found and only a limited set of SOAP specification was supported. For WS-Security which would enable message level security, there was no support at all.

RESTful Web Services basically require very limited support from the platform. As long as there is a programmable HTTP client API available with the support for the required HTTP methods it can be concluded that the platform supports RESTful Web Services. This is the case with iOS as documented in the iOS API specification [App15b]. Another interesting aspect is the support for the selected content type JSON. iOS contains a native JSON support [App15c] starting from iOS version 5.
Apple iOS contains support for the commonly used HMAC-SHA256 and HMAC-SHA512 functions [App06]. Other common security features are detailed in the security specification provided by the platform provider [App14]. Some of the relevant and interesting features in the context of this thesis include an API called Keychain. Keychain can be used to manage sensitive data such as passwords, encryption keys or X509 certificates. Each application has its own keychain and by default the user has to explicitly grant access to any other application that wants to access another keychain than its own. As previously explained, digital signatures backed by a proper PKI infrastructure can offer very strong security. However one of the key requirements is the secure storage of the private key. Multiple vulnerabilities have been discovered in the iOS keychain implementation [HeE12], although they have been rapidly patched in subsequent versions. One can question how trustworthy the keychain foundation is. This is typically the case for almost all software-based key stores. The keychain can certainly be used as a supplementary security solution, but it is questionable if it should be the cornerstone of the authentication and security solution.

The optimal storage for private keys is the Secure Element. Secure element is an independent hardware based tamperproof storage and execution environment that can be found for example from EMV chip based credit/debit cards or from the SIM cards provided by the mobile network operators. Apple devices which utilize the iOS operating system do not have a built in secure element which could be utilized by 3rd party applications running on top of the platform. The secure element found from iPhone 6 devices is currently exclusively meant to be used by the new Apple Pay payment scheme. Another interesting feature in the latest iOS devices is fingerprint recognition. In the initial release the fingerprint recognition is reserved for Apple proprietary solutions and it can not be utilized by 3rd party authentication solutions [App14].

5.2 Google Android

Android is a mobile operating system originally developed by Android Inc, acquired by Google in 2005. First Android version was launched in 2007. The operating system is based on the open source Linux kernel [Goo14]. Android is primarily targeted for touch screen devices such as mobile phones and tablets, but there are several other extensions such as Android-TV, Android Car and even digital cameras with the Android OS [Goo15e]. Android applications run on the Dalvik virtual machine, the programming
language is based on Java syntax, but due to licensing disagreements it is not officially Java [CNE14]. It does not fulfill the Java specification and only a subset of Java features is supported, and Android specific extensions have been added.

There is a significant difference between iOS and Android in how the operating system is being developed. While Apple has total control over iOS operating system and the hardware devices that run the operating system, Android is distributed as a plain vanilla version and hardware manufacturers are allowed to customize the operating system for their specific needs regarding functionality and branding. This has led to the situation where Android is significantly fragmented. After a new version of official Android operating system is published, it typically takes some time before the phone manufacturers have ported the changes to their customized version Android OS [Ars13]. As a consequence, critical security vulnerabilities that have been discovered and fixed in the main Android OS have been significantly delayed before being available for all of the modified Android versions [GoD14].

Remote communications support in Android closely resembles the situation with iOS. SOAP based Web Services are not natively supported by the platform as indicated by the Android API reference [Goo15f]. Third party libraries exist, but they contain limited support for the specification and especially WS-Security specification is not supported at all. On the other hand, RESTful Web Services are fully supported. There is an API for a HTTP client supporting different HTTP methods and JSON is natively supported by the operating system [Goo15b, Goo15c].

Android provides support for the commonly used HMAC functions similarly to Apple iOS [Goo15g]. Android 4.3 “Jelly Bean” release contains a feature called Android Key Store [Goo15d], which is the primary API that should be used for operating with keys. Interestingly the Key Store API also has built in support for Secure Element based key stores, which means that there is a truly secure way of managing key pairs in the Android on a built in hardware chip [Goo15d]. Unfortunately the number of devices containing the Secure Element is fairly limited. Another limitation is that there is a significant user base with older low end Android devices which only support older 2.X versions of Android. Therefore the benefits of using Secure Elements via the Key Store should be evaluated as it might limit the usage of the applications for significant number of Android users.
Latest high end Android devices such as Samsung S5 contain a built in fingerprint scanner. So far Android is lacking native support for fingerprint recognition [And14], but for example Samsung has opened their library for others to use [Poc14]. Fingerprint scanners will greatly improve the security of the end user device in case of theft, but so far there are not many applications which would use the fingerprint reader for actual user authentication. This might very well change in the near future.

5.3 Microsoft Windows Phone

Windows Phone is a mobile operating system by Microsoft Corporation. Windows Phone was originally launched in 2010 and it is the successor for the Windows Mobile. Original version was Windows Phone 7, it is a complete remake after Windows Mobile and the operating systems are incompatible regarding applications. Same incompatibly goes for versions 7 and 8.x within the Windows Phone family. Windows Phone 8.x is primarily targeted for mobile phones and tablets. For the tablets Microsoft also has Windows RT which is targeted for devices running ARMv7 based chipsets. Windows Phone and Windows RT are also incompatible which has caused some confusion in the Windows Apps space.

In terms of app development Windows Phone is more flexible than the competing Android and iOS based systems. Applications are running on Windows Phone Runtime, they can be developed with VB.NET or C#. Native Windows Runtime components can be developed in C++ or CX. There are multiple user interface solutions such as XNA or Windows Phone Studio. Microsoft also has a concept called Universal Apps, the applications conforming to this standard can run out-of-the-box either on Windows 8 or Windows Phone 8 based operating systems.

One could say that in terms of third party usage Windows Phone sits somewhere between iOS and Android. There are multiple device manufactures who utilize the operating system but the manufactures are not allowed to tailor the operating system for their specific needs as can be done with Android. So far Windows Phone has not gained significant market shares globally, in Europe and especially in the Nordics the situation is a bit better.

In the previous Windows Phone versions 7.x and 8.0 SOAP has been one of the supported communication protocols via the Windows Communication Foundation (WCF).
framework. However for the latest Windows Phone 8.1 version SOAP support has been deprecated from WCF for Windows Mobile. No official announcements have been done, but Microsoft employees have confirmed the discovery in the Microsoft developer forums [Mic15e]. This has caused some controversy in the Windows developer space as it yet another hiccup for the Universal applications when the integration capabilities between desktop and mobile do not match [Mic15e].

On the other hand, the removal of the SOAP support by Microsoft who is one of the key contributors of the original SOAP standard can be seen as a rather strong message stating they don’t believe in SOAP usage in the mobile environment. Similar message is distributed via Microsoft blog’s where the disappointed developers searching for the lost SOAP support are instructed to expose their end points as RESTful Web Services. This is a fair advice to those developers who have control over the end point; others unable to impact the service design are left disappointed.

As obvious based on the text above and confirmed by the Windows Phone API specification, RESTful Web Services and JSON are natively supported by Windows Phone [Mic15b] [Mic15c]. It contains a programmable HTTP client like iOS and Android as well and supports the needed HTTP methods. One distinct difference in the Windows Phone platform is related to the lack of API:s that can be used to hook in to the SSL handshake process [Cod14]. In iOS and Android it is possible to build additional validation logic into the SSL handshake to implement functionality called “Certificate Pinning” which is used to prevent advanced Man-In-The-Middle attacks based on forged CA certificates, in Windows Phone a third party HTTP client needs to be used to be able to achieve this functionality. Certificate pinning is elaborated more detail in chapter 5.5.

Windows Phone contains support for the commonly used HMAC-SHA256 and HMAC-512 functions [Mic15f]. Other interesting functionality related to key management is called Key Container for storing RSA keys and certificates [Mic15d]. In addition all of the basic functionality such as generating key pairs, signature generation and encryption are fully supported. So far there haven’t been any security bulletins regarding vulnerabilities in the Windows Phone key management area. This might be due to an extremely solid implementation, but one other rather likely reason is that so far there are no reports of successful jail breaking / rooting of Windows Phone based devices. As seen with iOS and Android most of the serious attacks require that the attacker manages to do a successful privilege escalation before the vulnerable API:s can be misused. Whether Win-
Windows Phone is designed to be more secure than iOS and Android, or is it only the lack of interest from the attackers since globally it is a rather marginal operating system, is still an open question.

Windows Phone supports the use of SIM card based Secure Elements which enables secure authentication, but the dependency towards the SIM card also means a dependency towards the operator. Latest Windows Phone version 8.1 also includes a feature called Trusted Platform Module (TPM) which usually referred as Secure Element. TPM enables the storage credentials and cryptographic keys for performing cryptographic operations (referred as Virtual Smart Cards by Microsoft) [Mic15a]. So far no Windows Phone based devices with finger print authentication have been published.

5.4 Common denominator in supported remote communication protocols

Based on the previous Sections 5.1, 5.2 and 5.3 it is obvious that all of the selected Mobile platforms have selected RESTful Web Services as preferred integration pattern. This is also clearly stated in the developer guidelines published by the platform vendors.

SOAP is only supported in previous versions of Windows Phone. All of the platforms support the fundamental pre-requirements for SOAP, meaning a HTTP client + XML support, but building a working client which conforms to the SOAP standard on top of the primitives will be an extensive task. Crafting the missing SOAP support is will decrease the productivity from the actual development work for a long time. Extending the homegrown SOAP client to support WS-Security for the best possible security solution changes the amount of work extensive to significant. This is also hinted by the fact that the open source SOAP frameworks for iOS and Android haven’t even published experimental versions with WS-Security support.

Security-wise the platforms do not offer many built in capabilities for the remote communications, any security solution going beyond simple SSL + HTTP basic authentication need to be handcrafted. For this purpose all of the platforms contain support for the commonly used HMAC functions with similar strengths. The support for HMAC functions enables message level protection with Hashed Message Authentication Codes.

One common security factor that can be utilized with all platforms is device registration (also called device authentication or device enrolment). This is especially relevant due
to the nature of the mobile, as explained in Section 2.6. The mobile device can automatically hop between networks or base stations to switch from Wi-Fi to mobile data and vice versa. This leads to a frequently changing IP address which means that client IP address is not a security factor that could be used for protection the integrity of an authenticated session and therefore it is relevant to find other means for protection against session theft.

Basically device authentication means that the app and device combination needs to be pre-registered by the customer before he can start to use the actual services (this is elaborated in more detail in Section 6.2). Device authentication can be combined both to user authentication and request authentication; for making sure that the customer is authenticating from a previously registered device and for making sure that all of the requests in a single session are originating from the same physical device.

All platforms contain functionality for storing sensitive keys in a key chain / key store, but the reliability of these mechanisms has been questioned by successful attacks both in Android and iOS platforms [Hay14, HeE12]. If X509 certificates are stored in the mobile device outside of the Secure Element it is a good precaution to limit the use of the certificates only within the mobile channel for specific services and not as a general purpose identity mechanism for any type of service. This is to limit the impact of a possible breach. Wider adaptation and standardization of Secure Elements and in mobile devices might very well be the key “enabler” which simplifies the development of secure and trusted mobile applications really to take off despite the increasing efforts of malicious attackers.

5.5 Other security related considerations for mobile platforms

There are several best practices and security improvements which can be used to harden the mobile applications against certain typical attacks. These measures and guidelines can be used to increase trust towards the device, limit the possibility of information leakage in the device, provide additional security for SSL protected communications and to make sure business logic is placed in the right layer so it can not be easily circumvented.

There is no definite way of protecting an application that has been published and is executed in a non-controlled environment, but following these recommendations will in-
crease the effort required to craft an attack against the mobile applications.

Certificate pinning is a technique where additional validation logic is built in to the SSL handshake. Core purpose of the validation logic is to limit the number of trusted Certificate Authorities (CA) [Wal14] by explicitly white listing the trusted SSL issuer(s) [OWA14]. By default there are tens or hundreds of trusted CA certificates including various different types of companies such as Hong Kong post in the platform trust store. And even worse, it is possible for users to manually install new trusted CA’s without truly understanding what is a CA and if it should be trusted or not.

The extra protection provided by certificate pinning is needed due to prevent a specific Man-in-The-Middle (MiTM) attack called “malicious WiFi access point attack”. For example, anyone can park a car outside of the Hilton hotel and create a WiFi access point called “Hilton Free WiFi”. This alone doesn’t get the attacker far. But if the attacker also creates a forged login page for the WiFi network that instructs the user to provide his email address for receiving a key for the network it will be possible to get a certain percentage of the users to install a malicious CA certificate. The attacker also needs to configure a “poisoned” Domain Name Service (DNS) to provide a forged end-point which routes all traffic through his endpoint. For example https://mobilebank.bank.com in the WiFi network. This example URL is then protected with a false SSL certificate which the platform would trust by default since the signing CA was added to the phone trust store by the when the user accepted to install the CA certificate in the email. This will enable a MiTM attack to break the SSL session and leave all data visible and potentially modifiable to the attacker unless there is message level protection in place. With certificate pinning implemented the attack is not possible since the attacker can not get the malicious CA in to the original hardcoded whitelist of trusted CA:s provided by the app publisher [Wal14].

Avoid local storage of sensitive data in the mobile device. Any data that is stored on the end user device is data that could be stolen. By storing the data outside of the mobile device, this attack vector is closed [OWA14]. Best practice is to store all sensitive data in the backend systems with multiple layers of protection before it can be accessed from Internet [ECB13]. When there is a mandatory business need to store data locally on the device, a good practice for protecting the locally stored data is to either store only server side encrypted information, or deliver encryption keys from backend server only after a successful multifactor authentication. Despite the fact that all of the mobile operating
systems come with a platform intended way of securely storing data, only one new critical security vulnerability need to be discovered and the data might be easily accessible for the attacker.

*Disable caching from platform libraries* some platform specific HTTP client libraries come with a default caching enabled. This might lead to undesired situations where sensitive data is cached by the platform. For example the HTTP client provided by the iOS platform has request caching enabled by default [App15e]. In the worst case scenario a HTTP post request containing user credentials is stored in the platform cache which might be harvested by malicious applications.

*Jailbreak detection* is a mechanism to detect if the user has installed a Jailbreak / Rooting kit on his phone to perform privilege escalation. While the use of a jailbroken phone is not a direct threat, it still exposes the user’s device for multiple operating system weaknesses and makes it a desirable target for malware, trojans and phishing apps. Depending on the use case the service provider should decide if jailbroken devices are completely blocked from the service, or if the information is just passed to the backend risk assessment engine where the final decision for accepting the transaction can be done [ECB13]. Although there are no definite numbers available, it appears that a significant portion of the attacks towards mobile applications require privilege escalation. By preventing the execution of the app on a jail broken device this threat is effectively mitigated. Still, it is good to remember that if there was a trustworthy way of identifying a jail broken phone, the operating system provider would most likely already be utilizing this mechanism in the operating system core.

*Obfuscation* of the app binaries is a mechanism of protecting the application source code and internal logic from attackers who want to decompile the application to understand its internal workings. Basically the obfuscator mangles the original binary before compilation, strips all variable names and replaces them with nonsense to make it difficult to understand what the program is doing. Although obfuscation makes it more difficult to understand the application, it is far from being effective. With enough effort it is possible to reverse engineer the obfuscated source. This also hints that anything that is published in to the open markets, will be opened, examined and exposed. So the security of the application must not be based on the assumption that the application could hide the security related logic (i.e. security by obscurity). As mentioned in the previous chapter regarding local storage, never leave an encryption key in to the application binary
and expect an obfuscator to protect that.

*Prevent debug output from default console log.* Debug logging is a development time activity which is needed to for various reasons, for example to print out stack traces, understand error root causes, verify that the application works as intended and detect details of unexpected activities. Before the app is published to the market places it is very important to completely disable debug logging, otherwise it will expose an easy to use attack vector. Default debug log might be shared (depending on mobile platform) and open to be read by any application. The worst case scenario would be an application printing user credentials to the debug log where they can be read by another malicious application without even violating the operating systems security policy.

*Build server-side mechanisms to block access from specific App version or mobile operating system versions.* Once the mobile app has been published and downloaded by the user there is no direct control left for the publisher. The application can be removed from the app store, but it can not be removed from the end user device. Therefore it is important to build sufficient server side controls to limit the access as desired. For example in the situations where a major vulnerability has been discovered from the published application, the publisher can block the usage of the app via the server side controls. Additionally the publisher should give instructions to the users to download the latest application version containing all necessary fixes. Similar control should be in place regarding the operating system version which is running the application. In case of severe operating system vulnerabilities the service provider should have the capability of blocking and informing users who are running the application on a vulnerable operating system.

*Avoid implementing authorization and critical business rules in the mobile apps.* Business logic should be implemented in the apps only for convenience reasons. All critical business and authorization rules should be implemented server side. As mentioned earlier in chapter 4.5 there is no way to build a perfectly protected mobile application that the server could trust. A service should provide the resources that the app is allowed to operate and all important business rules should be executed server side. Similar rules can be implemented and replicated in the mobile apps for convenience reasons. These reasons might for example relate to being able provide optimal user experience via instant feedback and guidance.
Implement device enrolment [ECB13]. Requiring the user to actively register the devices to be used with the services enables better monitoring and risk assessment to be performed. Registered device increases the strength of subsequent authentications since the information regarding the device provides an additional factor to the authentication (something that the user possesses). Device enrolment also provides additional security for traditional one factor authentication solutions such as username + password. These light authentication solutions might be used for viewing data or executing low risk operations. Stealing a static username and password will not be sufficient for authenticating to the Bank since the request has to come from a pre-registered device.

6 Suggested solution for offering secure Web Services for Mobile Applications

This chapter describes a logical architecture for providing secure Web Services for Mobile Applications based on the information and key findings presented in previous chapters. The chapter starts by introducing the conceptual high level architecture of how applications and systems are grouped and continues to drills down deeper in to the specific characteristics mobile applications and REST API architecture. Eventually the presented components are mapped in to a physical network and runtime environment. The reasoning for many of the architectural decisions is justified in the following sections which introduce the concrete security solutions built on top of the selected architecture.

After the architectural description of the solution the selected security and authentication related use cases are described and explained in more detail using UML sequence diagrams. Functional use cases such as “execute action” or “view information” are not described; the aim is to describe the general frame which enables all type of functional use cases to be implemented.

Certain features and capabilities which are the most relevant in the mobile and security context are explained and described in more detail, and the basic “client - server” Web architecture is given less focus. As this chapter provides a logical description of the architecture, it is not a detailed implementation guideline for an individual system, but a high level blue print that can be used as the foundation for multiple solutions requiring similar (security) characteristics.

---

8 As defined by The Open Group in Togaf 9.1 Enterprise architecture development framework, a logical architecture refers to an implementation-independent description of the architecture [Ope11]
Capabilities and requirements listed previous chapters are mapped into the presented solution to explain where they should implement to provide the benefits and protection described earlier. Last section in this chapter analyses the results achieved and evaluates the strengths and weaknesses of the suggested solution.

6.1 Architectural description of the solution

Based on the findings in previous chapters the decision is to use RESTful Web Services with JSON as the message format when creating services aimed for mobile service consumers. Most significant deciding factor for this recommendation is chapter five which describes the native support for different communication protocols in the mobile platforms. If the services are targeted to mobile consumers, they must be offered in a format that is convenient to use and supported by the mobile platforms. JSON is recommended due to less heavyweight processing and smallest bandwidth usage.

Following sections describe the architecture from different abstraction levels and viewpoints starting from a high level conceptual architecture and drilling down to the more specific mobile application and REST API architecture.

6.1.1 Conceptual architecture

The mobile channel consists of five major groups of applications and systems which are described below in Figure 6.1.1. These groups may contain several applications and systems which perform similar responsibilities within a certain limited context.

![Conceptual architecture of the mobile channel](image)

**Figure 6.1.1 Conceptual architecture of the mobile channel**

*Mobile Applications* are the end-user applications that can be downloaded from the app stores and installed in to the customer’s mobile device. These applications provide the user interface and associated application logic to communicate towards the Bank by consuming one or more RESTful Web Services.
REST APIs are responsible for providing a consistent RESTful facade which exposes to the internal core systems that provide the actual business functionality. This layer is also responsible for all of the channel specific security functions such as session establishment, request authentication including validation and coarse grained authorization.

SOA services are responsible for publishing an internal Web Services layer which provides access to core business applications regardless of the implementation or runtime environment of the Core Business Application. SOA services are typically realized in an Enterprise Service Bus (ESB), but may also contain individual services which fulfill the internal SOA requirements of the organization regarding self-containment, management, monitoring and common data types. Most of the subsequent sequence diagrams in following sections omit the SOA layer since in this context it only provides connectivity and adaptation to the core business applications without performing relevant business logic.

Core Business Applications provide the actual business functionality. They are shared between several different channels such as mobile, Web, Contact Centers, ATM’s and physical branch offices. Core Business applications contain authentication systems, account ledgers, payment systems and many more. They are a heterogeneous collection of applications running in various runtime environments and implemented in different programming languages.

Supporting services such as the online app stores, intrusion detection (or prevention) systems, audit logging, statistics gathering tools, runtime monitoring and management tools are utilized throughout the layers but they don’t have a significant role in shaping the architecture. Details regarding the supporting services are not described in more detail in the subsequent sections as they are not highly relevant in the context of building secure Web Services.

### 6.1.2 Mobile Application Architecture

This section describes the high level architecture for the mobile applications in a mobile platform agnostic way. The main goal of the architecture is to separate concerns so that the applications achieve a high degree of maintainability in the ever changing environment. Presentation layer, Business layer, Resource Access Layer and Local Data Sources are decoupled and abstracted so that changes in one area should not reflect to others. Security is a cross cutting concern and all components within the layered archi-
tecture might have security related responsibilities. Figure 6.1.2 describes the layers and some of the main components within these layers.

![Layered Mobile Application architecture](image)

**Presentation Layer:** Separate presentation layer components enable the redesign and change of UI design without touching components below in other layers. UI components handle the designed functionality of various use cases depending on the exact requirements and content included in the application. Additional input mechanisms, for example microphone used for speech recognition, a barcode reader or an image scanner utilizing the phone camera, might hook directly with some presentation layer UI components.

**Business Layer:** Separate business layer components enable the change of business rules without touching the components in other layers. Business components are business rule managers that handle business rules per use case. Session, state and authentication management as well as some helper and worker components may be considered to be part of the business layer as well.
Resource Access Layer contains Data Access Components which enable fetching data from different data sources without upper layer components needing to know where the data is served from. Data Access Components handle access to the app specific in-device and off-device data sources. Some resource access helper and worker components may be considered to be part of resource access layer as well. A dedicated Data Access Component is responsible for the communications towards the Bank REST API’s.

Local Data Sources contains volatile and persistently stored data that is stored locally in the mobile device. The data sources can be provided by the platform or created within the application. For example the platform specific key store is used for storing the secret keys. Session related data is cached in a volatile runtime cache provided by the application to avoid unnecessary network communications. Optional user preferences related to the application or any other data needed in offline-mode is stored in a persistent store. Sensitive data can be stored only in encrypted format and the encryption key must not reside within the device.

Security: Security is a cross-cutting layer. The components or the features of the Security layer are included and implemented in the relevant components of each layer. Some security solutions might cross all layers and might not require modification in other components. Certificate pinning and jailbreak detection elaborated in Section 5.5 are examples which belong to the Security layer.

6.1.3 REST API Architecture

This section describes the REST API architecture in a platform or framework agnostic manner for offering the secure Web Services. Key building blocks and functionalities are grouped as modules or packages which need to be implemented according to the principles defined by the selected RESTful framework.

In the simplest form a REST API consists of two pipelines, one for request and one for response. These pipelines then have associated actions which are executed step by step. As originally described by Mr. Fielding, a REST interaction resembles “pipes and filters” architecture style although it has two independent processing streams for the request and response [Fie00]. The data within a stream may be processed and transformed by several filters before it is dispatched forward to the corresponding backend destination or back to the customer. In this section the REST API applications are described
with pipes and filters based approach which enables the division of request and response pipelines in to multiple independent steps that need to be executed to achieve the desired functionality.

To enable strong decoupling between the user authentication solution and business services the REST API is divided in to two separate applications called “Authentication REST API” and “Business REST API”. Other reasons for creating two independent applications relate to the nature and life cycle of the applications. Authentication related services are expected to be fairly stable and have significantly slower release cycle while the business related services will be incrementally developed and new services and new versions of the existing services will be frequently published.

The division in two separate REST API applications can also be seen as a more future proof approach. When the next business case for a new application requiring a REST API emerges, there is a fully reusable authentication solution in place and the development can focus on developing a new business service for the desired functionality. Further reasoning for the division is explained in Sections 6.3 and 6.4 which describe the token based request authentication solution that supports multiple user authentication mechanisms.

Both of the REST API applications will contain several operations, typically one or more per use case. These operations will contain operation specific logic and common processing logic that is executed for all operations. Figures 6.1.3a and 6.1.3b describe the two REST API applications and present their request and response pipelines with the associated filters. Both applications share a lot of similar characteristics and responsibilities, but there is clear difference in them. Authentication REST API related operations are only used for initiating a session. Operations provided Business REST API require that a session has been previously established and that the request belongs to an existing session.
Authentication REST API Request Pipeline in Figure 6.1.3a contains the following filters:

Request parser is the first filter to be executed for all incoming request messages. It will read the incoming HTTP request and parse the HTTP headers, URL and the optional JSON payload into objects that the application can process. The parser must be hardened to resist Denial-Of-Service attacks and it should limit the maximum amount of memory used for processing an individual request. Hardening is necessary since a malicious request message might be huge in size to consume all memory of the server, have faulty JSON structure or contain embedded JavaScript or excessive nested structures.

HMAC Verification is an optional filter. It will be executed for authentication related operations after the initial device registration has been performed when there is a Hashed Message Authentication Code to be verified. Device registration and HMAC protection are elaborated in the next sections. For the device registration process no device authentication is performed, only user authentication towards the Core Authentication System will be done. This will be described in detail in Section 6.6.

App Lifecycle Management is a mandatory filter which is executed for all operations in the Authentication REST API. The filter will inspect User-Agent and custom HTTP headers provided by the Mobile Application to ensure that the application and mobile operating system versions are allowed to access the REST API. Vulnerable application or operating system versions can be blocked from the service for security reasons. The
filter can utilize both whitelisting for stating the explicitly accepted versions or blacklisting for explicitly rejected versions.

*Request validators* are operation specific filters which ensure that the request format including HTTP headers, URL and request payload and parameters are according to the business rules associated with the operation. Request validators will also detect and prevent SQL, XML and HTTP header injections.

*Request Transformers* are operation specific filters which will transform the incoming JSON based request to the format understood by the relevant SOA services that provide connectivity towards the Core Business Applications. This is typically a JSON to SOAP conversion but JSON can be transformed to other formats when required by the Core Business Applications.

*Backend Connectors* are operation specific filters which are responsible for routing and the actual communications towards the target system for the specific operation with the appropriate protocol. Backend connectors must implement service level monitoring and throttle the traffic towards the core systems according to predefined system specific limitations to limit the impact of DDoS attacks.

**Authentication REST API response pipeline** in Figure 6.1.3a contains the following filters:

*Response parsers* are operation specific filters with one-to-one match towards the backend connectors. Response parsers will parse the response received in to a format natively understood by the REST API application.

*Response transformers* are operation specific filters which will transform the received response message in to the JSON format to be returned to the mobile application according to the agreed interface specification. Response transformers may filter out excessive data or detailed error codes which are not relevant in the mobile context. Similarly sensitive information which should not leave the internal network might be filtered out or encrypted.

*Security Token creator* is an optional filter that can be used by several different user authentication related operations. Security Token Creator will create a JSON Web Token based security token according the definitions that are detailed later in Section 6.4 after a successful user authentication.
Business REST API request pipeline consists of the following filters:

Request parser filter has the same responsibilities as described with Authentication REST API for parsing the incoming HTTP request message.

Security Token Verification is a mandatory filter for all operations in Business REST API. It verifies that the request belongs to an established session. The filter will perform security token validation according to the description that will be introduced in Sections 6.3 and 6.4. The filter might make call-outs to external systems to verify that the token has not been revoked.

HMAC Verification is a mandatory filter for all operations. It will verify the HMAC signature covering the request message using the secret key embedded in the Session Security Token. This process will be elaborated in more detail in Section 6.5.

Request authorization is a common filter used by all operations to ensure that the customer is authorized to perform the activity he has requested. Request authorization filter will typically contact a remote read-only authorization database.

Request validators are operation specific filters which have the same validation responsibilities as described for Authentication REST API.

Request Transformers are operation specific filters which have the same responsibilities for transforming the request in to appropriate format for the related backend system as described for Authentication REST API.
Backend connectors are operation specific filters which have the same responsibilities for contacting appropriate backend systems as described for Authentication REST API.

Business REST API response pipeline contains the following filters:

Response parsers have the same responsibility of parsing the received response message in to a format natively understood by the REST API application as described for Authentication

Response transformers are operation specific filters which have the same generic responsibilities as described for Authentication REST API.

6.1.4 Physical architecture

The previously introduced application components are mapped in to a physical runtime environment in typical multilayered security oriented network architecture in Figure 6.1.4. The suggested solution is not limited to specific network architecture but can be adapted based on the existing network landscape.

Components are isolated in to designated network zones based on their usage and the relevant communication needs. The network zones are isolated by firewalls. Internal application layers including REST API's and SOA layer are load balanced and clustered. External load balancer is also responsible for the SSL connection termination and it is connected to an Intrusion Detection or an Intrusion Prevention System (IDS) or (IDP). IDS is a passive system that receives a broadcast of all of the network traffic that passes through the load balancer and it can trigger alerts based on preconfigured package signatures.

Runtime environments or operating systems for the applications are not specified. All modern server operating systems are able to host REST APIs implemented with various different tools. Vendors like IBM, Intel, Layer 7 and Vordel among others offer hardened appliance based server environments that are designed to operate as REST gateways within the DMZ network. Many of the products include hardware crypto acceleration and advanced threat protection capabilities making them worthy of a consideration.

Supporting tools which reside in the bottom of the figure inside the Management network are shared and used by multiple other applications in addition to Mobile Channel. They collect and receive data from application and infrastructure components in all layers of the network.
Figure 6.1.4 Physical (network) architecture with the primary infrastructure components and applications
The Common Log Repository on the bottom of the picture gathers audit log events from all customer interactions with all of the systems which participate in the end-to-end process. System management tools which also reside in the Management Network are used to operate and monitor the running production environment. The last supporting application is the Statistics gathering application. It will collect statistics from all relevant aspects of the application use as needed by the organization.

### 6.2 Mobile device and mobile app registration

All of the selected mobile platforms are focused on a single user paradigm. There is a single user account on the device and it is not possible change the user account without reinitializing the device. The fact that the device and specific the app in the device are intended to be used only by a single person opens up some new possibilities for improving the security of the authentication process.

This improvement can be achieved by requiring the customer to registering the device and app combination by using strong authentication. The reason why both device and the app need to be registered is related to the fact that in the future the Bank might release new applications for other purposes and it might be necessary to be able to distinguish for which specific applications the device has been authorized.

Device and app registration will be a prerequisite for taking the application in to use. A new identifier is generated for this combination in the registration process and used as an additional authentication factor in all subsequent communications. This feature also enables new use cases where the customer might be able to retrieve non-sensitive personal information only by launching the application and using the one factor device authentication to identify himself.

Figure 6.2 describes the enrolment process for device and app registration. The user is identified by using strong electronic authentication. An identifier for the device and app combination is generated and the service provider stores the identifier in to a device registry system for later use. The identifier will be retrieved from the device registry in subsequent logins to verify that the user is using the application in an authorized device.
Mobile Application = end user application installed in customers device
Authentication REST API = RESTful facade providing access to authentication related core services
Core Authentication System = A system responsible for authenticating the customer based on the credentials provided
Device Registry = A system holding information of the devices customer has registered for the service

Figure 6.2 enrolment flow

Following steps described in Figure 6.2 should be executed in the enrolment scenario:

1. User selects to enroll his device and application to be used with the services and performs a strong authentication. Strong authentication credentials and relevant information concerning the application are sent to the Authentication REST API.

2. Origin and version of the app is authenticated to make sure that this is the original app published and not a third party malware/phishing app and that the version is recent enough to be used.

3. Customer is identified by sending his/her credentials to the Core Authentication System.

4. Core Authentication System returns the confirmed user identity.

5. Device and app identifier is generated. It is an artificial unique identifier for the app and device combination.

6. User ID and device and app identifier are sent to the Device Registry system.
7. Success of the registration operation is confirmed.

8. Device and app identifier is returned to the mobile application. There might be several alternative flows regarding how the identifier is returned. In the simplest form it is sent over the wire with the response message. To enhance the security, the identifier could also be sent as SMS message, or achieve maximal security (and lowest possible user experience) it could be sent as a registered letter.

9. Mobile Application stores the identifier by using the appropriate secure storage space provided by the operating system, such as a key chain or a key store.

After the enrolment the mobile application will always provide this identifier, or a reference to the identifier, when the customer performs an authentication to use the application. The identifier is compared to the information stored in the Device Registry to verify that the user has authorized this device and app combination to be used.

There are two ways of delivering the identifier with the login request messages after registration: 1) attaching the identifier as plain text as part of the request for the service to compare the identifier to the one stored in the backend database or 2) calculating a cryptographic hash from the identifier and only delivering a reference to the identifier and the calculated hash sum over the wire to the server. If the server can recalculate the same hash value by using the stored device and app identifier referenced in the request, it means that the request is coming from an authenticated device.

Option 2 is clearly a more secure alternative and more complex as well since the identifier never leaves the device. Even if the SSL connection would be compromised there is no way to capture the identifier. Only a local attack targeted for the specific application in the mobile device could jeopardize the identifier.

Full device authentication scenario as a part of the standard user authentication flow will be described in Section 6.5, before that certain other relevant aspects of the authentication solution are elaborated in more detail.

Mobile devices are prone to loss and theft and therefore certain supporting services are needed to complement the solution. It is essential to offer a backup channel for example via phone based contact center or a Web based solutions for de-activating sold, lost or stolen devices. Enabling self service capabilities for the device management should lower the amount of customer service and support required.
By introducing the previously described device and app registration process an additional authentication factor has been established (something that the user possesses).

### 6.3 Token based security solution for session management

As identified in Section 4.3, a self-contained token based security solution is the natural replacement for server side session identifiers generally utilized by browser based Web applications. This section describes the generic characteristics of a token based solution before the next section applies the solution in practice. The token which is used as a proof of an authenticated session is called Session Security Token (SST) in this context.

In essence the SST is a replacement for the authenticated user id. It can also be seen as an authorization ticket proving that this user is authorized to access the services that require previous authentication. The token must be properly protected, contain a unique token-identifier, authenticated user identity, Time-To-Live (TTL) for the session, authentication mechanism used to obtain the token, information regarding who issued the token and how the token is protected. Additional information can be appended as necessary. If the same entity issues and later verifies to the token, information regarding keys and algorithms used may be omitted to save bandwidth used when passing the token over the network.

The SST shall not reveal the user id in clear text in case it would be captured from the wire, or from the running memory of the mobile device. It must be resistant to any tampering attempts and have a limited lifetime. Limited lifetime and tampering protection can be achieved by using a Time-To-Live timestamp and digital signature covering the entire token content. Appropriate TTL should be evaluated based on the service risk level. A financial service used for executing transactions might have a considerably lower TTL than an informational service which is to be used only for viewing data. Confidentiality can be achieved by encrypting the entire token, that way it is also safe for the client application to persistently store the token, at least for the duration of the session. Since the token will be sent to the service with every single request, the size should be minimized. Therefore it is important to select the optimal format for the mobile context, leave out unnecessary information and compress the token before encryption.
The token format is an agreement between the issuer and verifier. A typical established security token standard would be SAML 2.0 assertions which provide a wide range of expressive means to describe a security token [Can05]. A more modern and lightweight alternative is JSON Web Token (JWT) where the specification is still in draft level [JBS14]. JWT is a simple JSON based security token that can be digitally signed using JSON Web Signatures (JWS) which is also IETF standard draft [JBS15]. Since the token is going to be issued and verified by the same organization (and possibly even by the same application) the format selection is not a crucial issue. Selected token format will not be reflected outside of the REST API applications. JWT tokens are more verbose which fits well the mobile usage and since JSON is used as the messaging format for the actual Web Services it is the natural selection for the token format.

To adhere to best practices regarding session management there should be a possibility to revoke a previously issued token when the user does an active log-out before the session is set to expire [OWA13]. Active log out is one of the challenges of the token based approach. Since by default the token is self-describing and valid to the end of the TTL period there is no way to revoke a previously issued token without building additional logic and services for this purpose. Two primary alternatives exist; keep track of all issued tokens and invalidate them as requested or build a revocation list for the revoked tokens.

Keeping track of all issued token would defy the purpose of self-describing tokens and closely correspond to server side sessions. A more RESTful approach would be to build a revocation list that contains all of revoked the token-identifiers, following similar logic as a Certificate Revocation List in Public Key Infrastructure. Revoked token identifiers must be added to the token revocation list and one part of the token validation process must be to do a look-up to the token revocation list with the identifier of the token to be validated. If the token identifier is found from the revocation list, access must be rejected even if the TTL and signature of the token are valid. Since the tokens have a limited life time, they only need to be stored on the revocation list for the maximum duration of the token life time.

Furthermore the revocation list must not be stored on the REST gateway server performing the validation. Storing the list in the REST gateway servers would require synchronization of the list between the nodes, decrease scaling capabilities and be against the RESTful principles. Transient nature of revoked token identifiers suggests that a
memory cache based solution with extreme performance for read operations and which by default allows a predefined lifetime for the information, might be a good realization platform for the Token Revocation List.

Since the token has a predefined limited lifetime, the customer session is fixed in length by nature. A too short maximum session duration might be seen as an inconvenience and on the contrary issuing tokens with an exceptionally lifetime could be considered as a security issue. The solution is to offer tokens with a rather short Time-To-Live limit and offer a service for renewing the token when necessary. This renewal service should also utilize the token revocation list to ensure that the old token is blocked to prevent parallel sessions. The Mobile Application must implement local session management and renew the token as necessary based on user activity.

Use of cryptographically protected security tokens enables the services to be stateless regarding authentication and session management. The tradeoff is increased CPU usage as there are several cryptographic operations involved in the process (encryption, decryption and signature generation and validation) and increased request message size since the security token must be attached to every single request message. The SST is bound to be larger in size than a traditional session cookie used with most Web applications.

However CPU intensity is not a huge concern as it can be addressed in several ways. There is special hardware available which is targeted for extreme performance in cryptographic operations. Another possible approach to address the increased CPU usage is to utilize excellent scaling characteristics of the solution. There is no dependency between the issuer and verifier of the security token as long as they share the same cryptographic keys used in the process generating and the tokens. This enabling horizontal scaling of the solution by adding parallel processing nodes to the REST API layer hidden behind a load balancer.

6.4 Flexible solution for supporting multiple user authentication mechanisms

As described in Chapter 2 there is a requirement to be able to support multiple user authentication mechanisms. Ideally this will be achieved by separating the user authentication from request authentication. When user authentication and request authentication are decoupled there is no need for an individual service to know the details regarding
multiple supported user authentication mechanisms. Previously presented security token based approach is well suited to facilitate this requirement as will be demonstrated.

The solution is to divide the services into two categories; authentication services which are responsible for user authentication and business services which cover all the rest of the services which are used after authentication. These categories correspond to the Authentication REST API application and Business REST API applications presented previously.

Each authentication mechanism (one time codes, hardware security token or mobile application based) needs a dedicated authentication service in the Authentication REST API. The service must implement the specific flow related to the authentication mechanism. As a result of a successful authentication the identity of the customer is confirmed and a cryptographically protected Session Security Token is issued as described in the previous section. This SST is attached to all subsequent requests messages which are sent to the business services that reside in the Business REST API to prove that the request belongs to a valid session.

The combined request and user authentication process is illustrated in the following UML sequence diagram in Figure 6.4. Previously presented device registration and authentication is omitted from this diagram to illustrate the principle in a simplistic manner.
Figure 6.4 simplified login sequence diagram illustrating how user authentication (red dashed line) is decoupled from the business services (blue dashed line).

Following is a step by step explanation of the flow in Figure 6.4. To keep the illustration simple the SOA layer and error and exception flows have been omitted.

1. Customer initiates the login flow by providing the credentials required by the authentication mechanism by sending a request message to the Authentication REST API.

2. Authentication REST API transforms the authentication request into the format and protocol understood the Core Authentication System.

3. If the credentials provided by the customer are correct, confirmed user identity is returned from Core Authentication System.
4. Upon receiving the information of a successful authentication Authentication REST API will generate a Session Security Token according to the previous description containing the user identity, validity period (TTL) of the session and other user related information. The SST is then digitally signed for tampering protection, compressed to save bandwidth in subsequent transactions and encrypted for confidentiality.

5. SST is returned to the Mobile Application as base64 encoded string and can be now used in all subsequent request messages towards the Business REST API.

6. After successfully completing the authentication flow, the customer is logged in and can select to invoke any of the services provided by the mobile application. The basic principle is always the same, mobile application generates a request message and attaches the SST in to a HTTP header. In this example the customer selects ExecuteTransaction operation which sends a request towards the Business REST API. The request contains the session security token and parameters filled in by the customer.

7. Business REST API receives the request and extracts the Session Security Token from the HTTP headers. The SST is then decompressed and decrypted. After the token is in clear text the signature is verified and validity of the Time-To-Live limit is verified.

8. Business REST API does a look-up to the Token Revocation List to make sure that the token as not been revoked.

9. Token Revocation List returns information regarding the revocation status of the token. Once token status has been successfully validated, the request is authenticated. Identity of the customer is known and the flow may continue to next steps.

10. After the request has been authenticated Business REST API transforms the incoming request to a format and protocol understood by the Core Business System. The user identity extracted from the SST and the received parameters are inserted to the request which is sent to the appropriate destination.

11. Core Business System executes the transaction requested by the customer. Executing a transaction in this context can refer to any type of business operation
from fetching data to executing a change.

12. Business REST API receives the response from Core Business System and transforms it into JSON format.

13. Business REST API finalizes the flow by returning the response to the Mobile Application.

Presented service architecture provides a strong decoupling between user authentication and request authentication as described in the requirements in Chapter 2. There is no limitation on the number or types of supported user authentication mechanisms that can be supported with the described solution.

6.5 Message level protection with Hashed Message Authentication Codes

Previously described device and app registration and security token based session management are complemented with one more fundamental security related capability, Hashed Message Authentication Codes. Use of HMACs will guarantee message integrity even in the worst case scenario of a serious SSL vulnerability which would compromise confidentiality. It will also serve as an additional authentication factor in request and user authentication. As explained in Chapter 4.3 there are no standardized digital signature schemes available to be used with RESTful Web Services with JSON content, therefore a custom solution is required.

The Hashed Message Authentication is generated by the Mobile Application by executing a HMAC function as follows: digest=HMAC(request, key). The resulting digest is sent over the network with the original request. The service that receives the request will do the same calculation and compare if the resulting digest matches with the digest received in the request message.

In reality the process is a bit more complex than the previous description. As a prerequisite a key generation and sharing process is needed to share the secret key between the client and server. To ensure that the HMAC calculation produces identical results both in the client and server the request message must be normalized before the calculation. A canonicalization algorithm for normalizing the request must be executed. The canonicalization algorithm defines which parts of HTTP headers, request URL and request payload are included in to the calculation and how they must be ordered, encoded and
concatenated to generate one long string that represents the entire request. For validating the HMAC exactly same steps need to be taken to produce an identical result.

A single deviation in any part of the process executed by the client or the server will most likely result a deviation in the digests. As typical with HMACs, there is no output from error situations. The HMAC digests either match or do not match and there will be no additional information available of the reason why they do not match. Rigorous testing is required to make sure that the service and clients perform exactly same calculation logic in all scenarios. The benefit of the solution is the improved security of the end-to-end system as it contains an additional independent layer of security on top of transport layer security. However use of HMAC does not provide message level protection due to the strong coupling of the message and transport protocol in RESTful architectures.

Section 6.2 described a mobile device and app registration process. The process contains a common element with the suggested HMAC solution. In Section 6.2 the common element was referred as “device and app identifier”, and in this chapter as “secret key”. To simplify the solution and limit overhead of key management it is possible to use a single identifier/key for both purposes.

The only requirement for the format of the “device and app identifier” is that it is unique by a reasonable high probability. Requirement for the secret key used with the HMAC is that it is unique by a reasonable high probability and that in addition the minimum key length should match the block size of the selected HMAC algorithm. Commonly used HMAC-SHA algorithms also accept keys that are longer than the block size, and there is no upper limit for the length of “device and app identifier” so clearly these two can be combined regardless of the block size of the selected HMAC algorithm.

In Figure 6.5 on the following page the HMAC solution is combined with the previously presented security token and device authentication approaches.
Mobile Application = end user application installed in customers device
Authentication REST API = RESTful facade providing access to authentication related core services
Business REST API = RESTful facade providing access to all core business applications except authentication
Token Revocation List = A system that holds a list of all of the revoked Session Security Token
LDAP Authorization = A system holding authorization data that describes which of the operations/services/assets the user is authorized to use
Core Authentication System = A system responsible for authenticating the customer based on the credentials provided
DeviceRegistry = A system holding the user specific secret keys that are used for HMAC signatures
Core Business System = A system (or collection of systems) responsible for providing the actual business functionality

Figure 6.5 Full user authentication and request authentication flows with HMAC protection.

HMAC generation and validation process requires following steps described in Figure 6.5:

0. As a prerequisite, the mobile app must have a secret key which is also known by the server which will verify the Hashed Message Authentication Code. The enrolment process to obtain the secret key was described in Section 6.2.

1. User selects to login into the application and provides his credentials for the selected authentication mechanism. The mobile app generates a RESTful request with JSON payload containing the credentials. Request goes through a canonicalization process to normalize and concatenate all request content including HTTP headers, request URL, request payload (if present) in to a single string.
A SHA256 HMAC digest is calculated from the canonicalized request with customer’s secret key. The key is stored by the mobile application in the platform specific key container provided by the operating system. The resulting digest is attached to the HTTP headers of the request with a reference to the key used. The request is now ready to be sent to the service.

2. The request containing user’s credentials, calculated HMAC digest and the identifier for the key used is sent to the Authentication REST API to a service responsible for the selected authentication method.

3. After receiving the request Authentication REST API transforms the request into appropriate format and calls the core authentication system to verify user’s identity.

4. Verified user identity is returned from Core Authentication System.

5. Authentication REST API creates a new request towards the Device Registry to fetch the associated secret key. The request contains the verified user identity from the Core Authentication System and the secret key identifier (HmacKeyId) from the original request.

6. Device Registry returns the user’s secret key as the response.

7. Authentication REST API will execute exactly same canonicalization process for the received request as was done by the Mobile Application in Step 1 to create a single string representing the entire request\(^9\). Authentication REST API will calculate a SHA256 HMAC digest from the canonicalized request and compare it to the HMAC digest received from the customer. If the digests match, the request is coming from an authorized device holding the correct secret key.

8. Authentication REST API will generate a Session Security Token which contains the confirmed user identity, Time-To-Live (i.e. the validity period of the session), authentication mechanism used and the user’s secret key for subsequent use. Session Security Token is digitally signed, compressed and encrypted.

---

\(^9\) Extra care must be taken to make sure that only the original http headers are taken into the calculation and that they are used in the same exact order as in when signing the request. HTTP 1.1/RFC2616 does not guarantee that the original order of HTTP headers is preserved. Various network nodes such as proxy servers typically inject additional headers to the request while in transit. Any deviation to the original canonicalization process regardless related to content or order of elements will cause the HMAC comparison to fail.
9. Session Security Token is returned to the Mobile Application and the customer is now logged in.

10. User selects to execute specific functionality within the application. Mobile Application will generate a RESTful request, and a HMAC digest covering the request message by using the secret key stored in the platform specific key container. Digest and the previously received Session Security Token will be attached to the HTTP headers of the request.

11. Mobile Application sends the request to Business Service REST API.

12. Business REST API verifies the delivered security token by decrypting, decompressing, validating the digital signature and by checking that the TTL indicates that the token has not expired. Business REST API also extracts the confirmed user identity and secret key from the Session Security Token.

13. Business REST API verifies the HMAC covering the request message by executing the previously described canonicalization and HMAC calculation process.

14. Business REST API invokes IsTokenActive service from the Token Revocation List system to ensure that the token has not been revoked.

15. Token Revocation List returns the information regarding token status (VALID/REVOKED).

16. If the security token and HMAC verification processes succeed Business REST API will query the LDAP Authorization database to ensure that the customer is authorized to perform the operation he has initiated.

17. Business REST API receives response for the authorization request.

18. If the customer is authorized to perform the operation Business REST API will call the Core Business System to execute the functionality requested by the customer.

19. Core Business System will execute the requested functionality.

20. Core Business System returns the response of the operation to Business REST API.

21. Business REST API converts the received response message into JSON format and returns the response to Mobile Application.
The sequence described in Figure 6.2 contains a multiple of steps and there are several systems involved. This might cause some concerns regarding the performance and response times of the solution. It is important to notice that user authentication is the more heavy weight operation where several remote calls are needed and the request authentication only contains simple external lookup operations. These lookups are used to check the token revocation list and to verify if the user is authorized for the operation. User authentication happens only once for each session while request authentication is performed for every single service call after user authentication. This justifies the more heavyweight processing during user authentication.

The HMAC based authentication fits well in to the proposed stateless service architecture. In essence the HMAC provides two benefits, it proves that the customer is in possession of a registered device and it also protects message integrity in case of SSL vulnerabilities. The solution does not conflict with either the device registration or the security token solution for session management but complements them.

### 6.6 Mapping solutions to requirements

This chapter is a summary of the various requirements presented throughout the thesis and corresponding architectural or technical solutions to address the requirement. Table 6.6 gathers the requirements and presents the solutions. “From chapter” in the table indicates where the requirement has originated from and “In Chapter” refers to the chapter that describes how the requirement is fulfilled.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>From Chapter</th>
<th>Solution</th>
<th>In Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1: solution must scale horizontally for a rapidly increasing user base</td>
<td>2.2</td>
<td>Stateless services, security token approach. Local caching in mobile applications.</td>
<td>6.1, 6.3</td>
</tr>
<tr>
<td>R2: Multiple strong authentication solutions must be supported</td>
<td>2.6</td>
<td>Authentication service concept where a user authentication flow produces a “standard” security token. No limitation regarding the number of possible strong authentication solutions that could be integrated to the system.</td>
<td>6.4</td>
</tr>
<tr>
<td>Requirement</td>
<td>From Chapter</td>
<td>Solution</td>
<td>In Chapter</td>
</tr>
<tr>
<td>----------------------------------------------------------------------------</td>
<td>--------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>R3: Selected remote communication protocol is supported by the selected mobile platforms</td>
<td>2.1</td>
<td>RESTful Web Services with JSON supported by all of the selected mobile platforms.</td>
<td>5.4</td>
</tr>
<tr>
<td>R4: Security solution supported in all mobile platforms</td>
<td>2.1</td>
<td>All platforms have a secure storage for storing cryptographic keys. All platforms support Hashed Message Authentication Codes.</td>
<td>5.4</td>
</tr>
<tr>
<td>R5: Solution must fit in to the existing enterprise architecture</td>
<td>2.6</td>
<td>A new mobile channel is created to access the existing core business applications</td>
<td>6.1</td>
</tr>
<tr>
<td>R6: Existing core systems should not be modified</td>
<td>2.6</td>
<td>Existing core systems will be exposed AS-IS via RESTful facades. Only new potential core system is the customer mobile device + app registry which could potentially be reused by other applications with mobile related needs (such as Push notifications).</td>
<td>6.1</td>
</tr>
<tr>
<td>R7: Implement authentication</td>
<td>2.2</td>
<td>Multiple strong authentication solutions supported, separate protocol for request authentication</td>
<td>6.1</td>
</tr>
<tr>
<td>R8: Implement authorization</td>
<td>2.2</td>
<td>Authorization must be performed by the core business systems and can optionally be performed also in the Mobile channel by propagating the authorization information to a LDAP database</td>
<td>6.1</td>
</tr>
<tr>
<td>R9: Implement auditing</td>
<td>2.2</td>
<td>All meaningful events are stored to a common log repository</td>
<td>6.1</td>
</tr>
<tr>
<td>R10: Enforce message integrity</td>
<td>2.2</td>
<td>HMAC which protects the RESTful messages</td>
<td>6.5</td>
</tr>
<tr>
<td>R11: Enforce message confidentiality</td>
<td>2.2</td>
<td>Use SSL for encrypting the message exchange on transport layer</td>
<td>6.1.4</td>
</tr>
<tr>
<td>R12: Address mobile constraints (weak cpu, limited memory, limited bandwidth, changing IP address)</td>
<td>2.7</td>
<td>RESTful Web Services with JSON is the most lightweight approach for remote communications. Session theft is not limited by fixating to an IP but based on the HMAC signatures where stolen messages only contain a signature without credentials to reproduce on another message.</td>
<td>3.4</td>
</tr>
<tr>
<td>Requirement</td>
<td>From Chapter</td>
<td>Solution</td>
<td>In Chapter</td>
</tr>
<tr>
<td>-------------------------------------------------------</td>
<td>--------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>R13: Require customers to register the mobile device they use for the service</td>
<td>2.5</td>
<td>Mobile device + app registration by using strong electronic authentication requires that the customer authorizes the device where she/he intends to use the application.</td>
<td>6.2</td>
</tr>
<tr>
<td>R14: adhere to relevant legislation and regulation</td>
<td>2.5</td>
<td>A broad topic that needs to be analyzed with Subject Matter Experts when creating the detailed implementation design. Out of scope for this thesis.</td>
<td>-</td>
</tr>
<tr>
<td>R15: the service must be able to block old or vulnerable mobile app or mobile operating system versions</td>
<td>5.5</td>
<td>App Lifecycle Management filter inspects the HTTP headers provided by the mobile application and grants or denies access by comparing those to a predefined configuration (white listing allowed or black listing blocked version)</td>
<td>6.1.3</td>
</tr>
<tr>
<td>R16: mobile channel must throttle traffic so that the internal core systems are not disturbed in a DDoS attack.</td>
<td>4.5</td>
<td>Backend connectors responsible for communicating towards the internal core systems must implement service level monitoring and throttle traffic according to predefined limitations.</td>
<td>6.1.3</td>
</tr>
<tr>
<td>R17: address all OWASP TOP 10 mobile threats.</td>
<td>2.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OWASP M1: Weak server side controls</td>
<td>2.3</td>
<td>Hardened REST API. All functionality behind a strong electronic authentication. Security token and HMAC signatures for request authentication.</td>
<td>6.1.3, 6.5</td>
</tr>
<tr>
<td>OWASP M2: Insecure data storage</td>
<td>2.3 (2.5)</td>
<td>Avoid local storage of data in the mobile device. Only use protected storage capabilities of the mobile platforms when necessary. Jailbreak protection to limit the application from running in a compromised platform.</td>
<td>5.5</td>
</tr>
<tr>
<td>OWASP M3 Insufficient transport layer protection</td>
<td>2.3</td>
<td>Always use SSL for communications, implement certificate pinning for additional security</td>
<td>6.1, 5.5</td>
</tr>
<tr>
<td>OWASP M4: Unintended data leakage</td>
<td>2.3</td>
<td>Prevent debug log output in mobile apps. Disable platform specific default cache mechanisms to avoid data leakage.</td>
<td>5.5</td>
</tr>
<tr>
<td>Requirement</td>
<td>From Chapter</td>
<td>Solution</td>
<td>In Chapter</td>
</tr>
<tr>
<td>---------------------------------------------------------</td>
<td>--------------</td>
<td>---------------------------------------------------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>OWASP M5: Poor Authorization and authentication</td>
<td>2.3</td>
<td>Utilize proven strong electronic authentication mechanisms. Never implement local authentication in the mobile applications. Leave authorization decisions to server-side controls.</td>
<td>4.5, 6.1, 6.5</td>
</tr>
<tr>
<td>OWASP M6: Broken cryptography</td>
<td>2.3</td>
<td>Review implementation with security experts, perform penetration testing and verify that all (encryption and signature) algorithms are considered to be modern and secure.</td>
<td>-</td>
</tr>
<tr>
<td>OWASP M7: Client side injection</td>
<td>2.3</td>
<td>Input validation should be performed to all input received by the mobile application (either coming from the user or reading a local database). Not addressed in this thesis.</td>
<td>-</td>
</tr>
<tr>
<td>OWASP M8: Security decisions via untrusted inputs</td>
<td>2.3</td>
<td>Related to inter process communications within the mobile platforms. Not addressed in this thesis.</td>
<td>-</td>
</tr>
<tr>
<td>OWASP M9: Improper session handling</td>
<td>2.3</td>
<td>Use of HMAC signatures prevents session stealing. The secret that is used for signature generation is never transported over the wire. Session security token approach enabled more fine grained controls for sessions and also enables active logout with the Token Revocation List.</td>
<td>6.1.3, 6.3, 6.4</td>
</tr>
<tr>
<td>OWASP M10: lack of binary protection</td>
<td>2.3</td>
<td>Obfuscation of source code before compilation makes reverse engineering more difficult but not impossible. Jailbreak detection helps to enforce that the application is only executed in a safe environment.</td>
<td>5.5</td>
</tr>
</tbody>
</table>

Table 6.10 Mapping solutions to the presented requirements

6.7 Analyzing the strengths and weaknesses of the solution

The described solution which utilizes RESTful Web Services with JSON as the integration protocol could be described as the expected end result. SOAP based Web Services in the mobile context seem to be a rarity. Although technically nothing would prevent handcrafting the needed support, financially it wouldn’t be worth the effort.

The token based security solution which enables any kind of credentials to be ex-
changed for a standard token is a clever replacement for the server side sessions but it comes with the tradeoffs of increased request message size and additional server side processing. Heavy use of cryptography increases the need of the server side processing which can be mitigated by utilizing the scaling characteristics of solution or by using hardware crypto accelerators.

Cryptographic processing will most likely slightly increase latency of the services, but still the most significant source of latency is the mobile network. The presented solution has succeeded to keep the invocations from the mobile app to the Web Services at a minimum level. Only in the initial authentication flow there is a need to perform two distinct operation at the same time, user authentication and fetching of data for the first screen after login. The security related use-cases described in the sequence diagrams contain several internal remote calls after receiving the request from the Mobile Application. Those invocations happen within the internal network of the Bank and are not expected to introduce substantial latency as the Token Revocation List and the LDAP authorization database are located in the same network zone as the REST API. The operations performed towards these systems are lightweight lookup operations typically returning a boolean or a single string.

Hashed Message Authentication Code provide a strong solution for request authentication and request integrity protection. Embedding the secret key used for HMAC calculation inside of the Session Security Token fulfills the stateless requirement when the token and the message signature can be verified independently without maintaining server side state.

The described security solution consists of independent building blocks: token based session management, device registration and Hashed Message Authentication Codes. All of the building blocks were used to craft a solution with very high security capabilities, however they work independently as well. Based on the detailed requirements it would be possible to cherry pick only the mechanisms relevant for a particular solution.

Ability for performing the active log-out requires a server side state to be maintained for the revoked tokens which could be seen as a violation of RESTful principles. The token revocation list also brings additional complexity to the solution. RESTful principles regarding caching will also be violated with the indirect object references described in Section 4.5. However since the sensitive nature of the information processed does not
allow any network level caching this can be seen as a minor issue.

The described security solution offers strong protection for the communications between customer and the Bank, but provides limited protection for local attacks. Following the mobile platform provider’s recommendations for building secure applications and extensive penetration testing are the best ways of enforcing local security of the apps. Making sure that the customer is using a non-compromised and recent enough version of the mobile operating system combined with the means described in chapter 5.5 also provide a good mitigation scheme for the local attacks.

Inventing new security solutions is not an encouraged best practice, quite the opposite. However the cost of developing and maintaining a WS-Security based solution would have been so high that it was decided that inventing a new security solution based on examples shown by others was feasible in this case to be able to utilize the benefits offered by RESTful Web Services.

When comparing the presented solution with traditional secure Web applications there are some clear security improvements. These improvements include as Man in The Middle protection for SSL via certificate pinning, request authentication and integrity protection with Hashed Message Authentication Codes and authorization of the end user devices.

7 Conclusions

This thesis has examined the different aspects related to offering secure services for mobile applications. Different axis in the research have related to the communication protocols, security solutions and support within the selected mobile platforms. The end-result is a RESTful architecture which is able to fulfil the requirements originating from several different sources. RESTful Web Services with JSON was the natural selection for the integration protocol as it is natively supported by the selected mobile platforms and JSON requires least heavy processing by the mobile devices.

While the stateless nature of the RESTful Web Services has generated some interesting challenges, the thesis presents a token based solution to which enables the client application to hold the session state. One specific challenge remained, if the applications must implement an active logout scenario, the services have to somehow maintain the state of the revoked tokens. While this could be seen as violation of the RESTful princi-
ples, looking at the big picture it is a rather minor issue which can be solved with presented solution involving a token revocation list.

Although RESTful Web Services currently lack formally standardized security solutions, a modular solution was constructed partially based on examples and best practices utilized by others. The strength of the modular solution is in the independent building blocks. It is possible to utilize the HMAC signature scheme without device registration or vice versa, and the session security token solution can be used to complement either or both of these approaches. Final decision regarding the security solution should always be based on the actual business requirements; more security comes with more complexity and cost.

The research did not challenge Dr Fielding’s doctoral dissertation regarding where to store the authentication state of the session. However since the target of Fielding’s dissertation was more focused on open and highly scalable Web Services it might be an interesting area for future research to investigate if the same really holds for “closed” Web Services where all services require previous authentication. It might be possible to replace the token based approach with a cookie based identifiers and still maintain the stateless benefits. This would require that the session management would be established as an independent core service and removed from the Web Service layer.
References

App06  Apple Inc, iOS HMAC API documentation, 2006,  

App14  Apple Inc, iOS Security, 2014,  

App15a  Apple Inc, App Distribution Guide, 2015,  

App15b  Apple Inc, NSURLConnection API documentation, 2015,  

App15c  Apple Inc, NSJSONSerialization API documentation, 2015,  

App15d  Apple Inc, iOS API library, 2015,  

App15e  Apple Inc, NSURLCache API documentation, 2015,  

And14  AndroidAuthority.com, Source code reveals discarded Nexus 6 Fingerprint Scanner, 2014,  


Fer07 | Ferg, B. et al., OpenID Authentication 2.0 – Final, 2007, [http://openid.net/specs/openid-authentication-2_0.html](http://openid.net/specs/openid-authentication-2_0.html) [28.8.2014]

Fielding, R., Architectural Styles and the Design of Network-based Software Architectures, 2000,

Finextra Research, Banking Trojan hijacks out-of-band SMS security, 2011,

Finlex, Laki vahvasta sähköisestä tunnistamisesta ja sähköisistä allekirjoituksista (Legislation regarding strong electronic identification and digital signatures in Finland), 2009,

Finansiell ID-Teknik, Mobile BankID statistik, 2013,


Google Inc, Building Kernels, 2014,

Google Inc, Android Security, 2015,


Hay14  Hay, R., Android key store vulnerability, IBM, 2014,
http://securityintelligence.com/android-keystore-stack-buffer-overflow-to-
keep-things-simple-buffers-are-always-larger-than-needed/ [1.9.2014]

HeE12  Heider, J., El Khayari, R., iOS Keychain Weakness FAQ, 2012,
https://www.sit.fraunhofer.de/fileadmin/dokumente/sonstiges/iPhone_keych
ain_faq.pdf [18.9.2014]

IBM12  IBM Corporation, Tatanga Trojan Bypasses Mobile Security to Steal Money
From Online Banking Users in Germany, 2012,
http://securityintelligence.com/tatanga-trojan-bypasses-mobile-security-to-
steal-money-from-online-banking-users-in-germany/#.VMo382OEPrs
[29.1.2015]

JBS14  Jones, M.; Bradley, J.; Sakimura, N.; JSON Web Token (JWT), IETF draft
[16.2.2014]

JBS15  Jones, M., Bradley, J., Sakimura, N., JSON Web Signature (JWS), IETF
signature-41 [18.2.2015]

Jut15  Jutila, J., Tunnistuksen “täydellinen myrsky” tulossa Suomeen (Perfect
storm of authentication about to hit Finland), TIEKE (Finnish Information
Society Development Centre), 2015,

Kra97  Krawczyk, H. et al, HMAC: Keyed-Hashing for Message Authentication,
[10.2.2015]

Lad11  Ladan, M.I., Web services: Security challenges, Internet Security
2011

Law06  Lawrence, K. et al., Web Services Security: SOAP Message Security 1.1,
2006, OASIS standard, https://www.oasis-
open.org/committees/download.php/16790/wss-v1.1-spec-os-
SOAPMessageSecurity.pdf [18.2.2015]


<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
<th>Authors/Details</th>
</tr>
</thead>
</table>
Tiv12 Tietoviikko (Finnish ICT News Week, weekly magazine), Suomalaisiin verkkopankkeihin hyökättiin (Finnish Internet Banks under attack), 2012

Tiv15 Tietoviikko (Finnish ICT News Week, weekly magazine), Miksi verkkohyökkäys jumitti myös pankkiautomaatit, OP? (OP why did the DDoS attack also disable ATM:s?),

