RoboBank
Technical Approach

STATUS: ROUGH

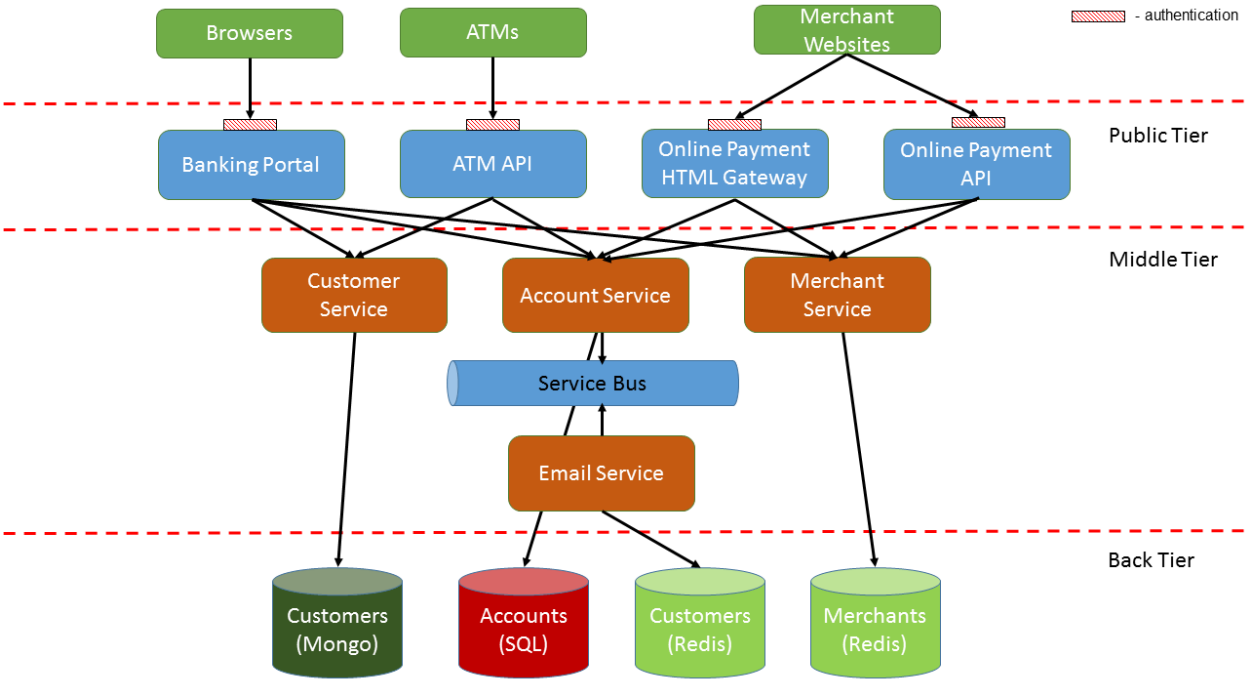
AUTHOR: ADRIAN BONTEA

System Architecture

Relevant Patterns, Principles and Practices

The following patterns, principles and practices are relevant in the context of RoboBank system architecture: N-Tier Architecture, Microservices, EDA,NoSQL, Cloud Computing.

Applications View



The system is composed of 9 applications (together with the application running on the ATMs) distributed in 2 tiers: Public and Middle.

The system architecture follows the Microservices style by including some relatively small services each running in its own process and communicating with lightweight mechanisms. These services are organized around business capabilities (in essence applying SRP at a system level) and are independently deployable and scalable. The approach involves a decentralization especially at a data level:

Model decentralization: Each service can maintain its own view of the same data as appropriate for that (bounded) context. E.g. Customer concept is represented differently in the Customer Service and Email Service contexts although the identity refers to the same person in the real world. In some cases, this can involve some amount of data duplication which is perfectly acceptable for flexibility (decoupling) and performance reasons.

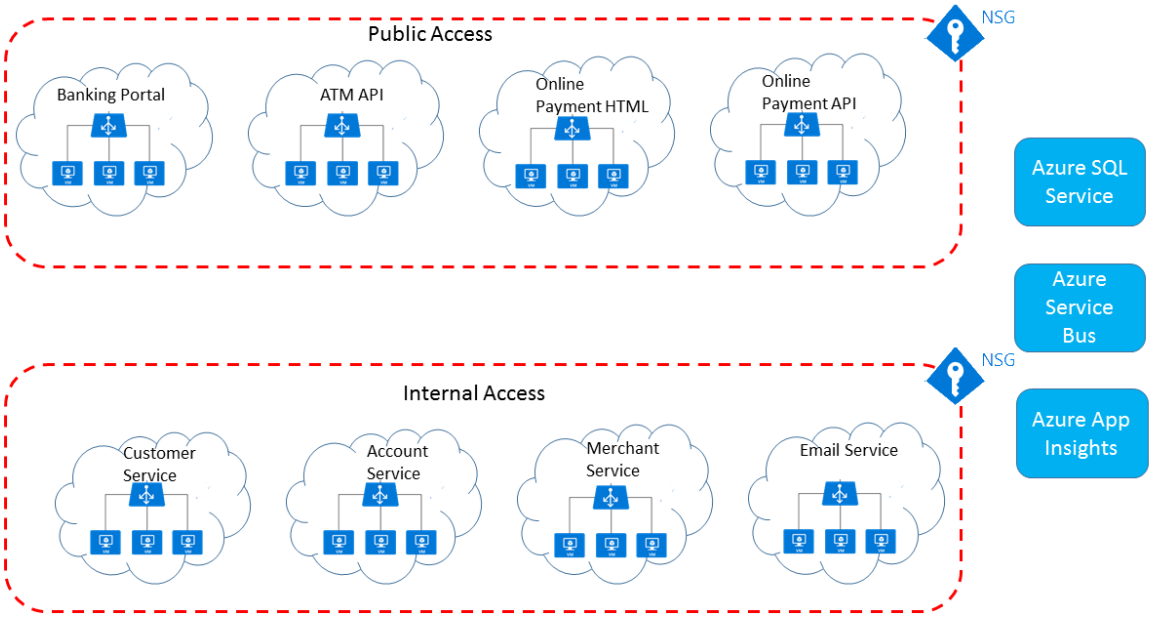
Technology decentralization: Each service manages its own database, either as separate instances of the same database technology or entirely different database systems. (Polyglot persistence) This allows each service to use the tools that are appropriate for the job.

E.g. Customer service uses Mongo because is horizontally scalable (sharding and replication) and schema-less while allowing queries by custom fields. This enables the implementation of some features such as having custom fields added by customers on their profile (profile data differing between natural and legal persons and even between the same kind of persons) and searching for that fields.

Account service needs a RDBMS since it needs support for transactions across different tables or rows in the same table for funds transfer between two accounts.

Email Service and Merchant Service are using Redis since it is a very fast (in memory) and horizontally scalable (Redis cluster) key-data structure (strings, lists, sets hashes etc.) store with persistence support (RDB or AOF) which makes it appropriate for storing the kind of data that these two services need. Still, the services can use separate instances (single node each or separate clusters) of Redis.

Infrastructure View



The system can be deployed entirely in Azure, on premise (requiring some small adjustments to Email Service implementation to extract the logic consuming the Azure Service Bus Queue from the custom Azure-specific role entry point into a generic class library to be reused in a windows service e.g.) or follow a hybrid deployment model. On the on premise model the system can still make use of some very helpful and cheap Azure services such as Application Insights and Azure Service Bus. (Currently functional when running the entire system locally in Azure compute emulator)

Deploying each application as a separate cloud service provides flexibility to development teams that can independently develop and publish their services. A scenario involving the applications deployed as a separate web/worker role in the same cloud service involves some overhead since a role cannot be currently deployed (all instances) independently of the other roles in the same cloud service unless the application is deployed separately on each role instance. (VM)

On the middle tier security is provided by infrastructure: In Azure this basically means putting all Middle tier role instances from all cloud services in the same subnet and defining a Network Security Group with strict rules (only allow traffic from Public tier) over this subnet. This same scenario can be realized on premise using a firewall.

On the public tier security is provided at an application level by having each application using the authorization protocols that are appropriate for the context – more details below.

More, all traffic to Public tier should be restricted to TLS/SSL.

Implementation Details

Relevant Patterns, Principles and Practices

The following patterns, principles and practices are relevant in the context of RoboBank system implementation: DDD, BDD, TDD, Clean Architecture/Onion Architecture/Hexagonal Architecture/Cone Architecture.

Middle Tier

The implementation of the services in this tier involves two separate VS solutions sharing some projects according to the structure involved by Hexagonal/Onion/Clean/Cone Architecture patterns:

The RoboBank.X.Core solution includes the “high level” projects (policies of the application, the general abstractions that govern the entire application; the truths that do not vary when the details are changed). These projects only reference other projects within the core or .NET BCL assemblies. (Dependency Rule: Source code dependencies can only point inwards) This approach provides a pure reusability of the core in different context while also decreasing the rigidity as compared to traditional Layers pattern (interfaces are used to express services that the core needs hence are more stable than the leaky abstractions provided in the “Infrastructure/Cross-Cutting” layer implementation in a traditional Layers pattern)

The RoboBank.X.Service solution shares the same core projects and adds Adapters projects (the glue that connects the application core to the details that support it. Adapters are user code implementations that are specific to various details and that hide these away from the inner core circles)

All these services are using ASP.NET WebAPI 2 as the delivery technology together with Swagger for API documentation. (a better looking and feeling alternative to ASP.NET WebAPI help page)

Exception handling is provided at the delivery technology level: using custom WebAPI exception filters that are logging the exception using ELMAH and modifying the response to include just a summary of the exception (exception message as opposed to the full stack trace that is returned by the default ASP.NET WebAPI filter for uncaught exceptions)

ELMAH is currently configured at a minimal level and does not include persistence configuration. (All exceptions are being logged in memory and are available on the hostname/elmah.axd route only during the lifetime of the host process)

Customer Service

Account Service

Merchant Service

Email Service

Public Tier

Banking Portal

[Not yet implemented]

This should be a web application providing some basic E-Banking functionality: funds transfer, accounts balance info etc.

A more special kind of functionality should be provided by this application to legal person customers: the ability to define website keys and secret keys to be used for integrating their merchant websites with RoboBank online payment – more details below.

The functionality should be delivered using ASP.NET MVC together with Bootstrap framework for responsive design.

Authentication should be implemented as a claim-based approach using LinkedIn Oauth Provider.

ATM API

[Not yet implemented]

This application should expose a Web API to be consumed by RoboBank apps running on ATMs.

Authentication should be implemented as a custom HMAC approach similar to what Buckaroo JSON Gateway requires:

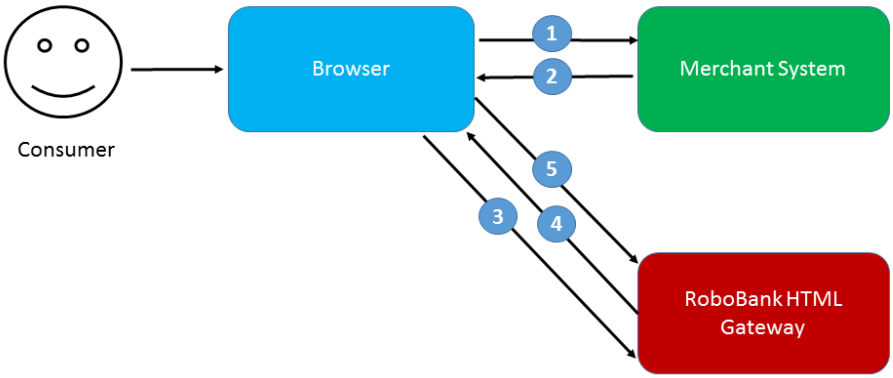
For every request the client is required to pass a custom format token in HTTP Authorization header. This token includes the “hmac” sequence, a Base64 hash using HMAC SHA256 signed with a secret key that both ATM API and the ATM app know, a nonce (random sequence of characters that should differ from for each request) and a request timestamp. The HMAC SHA265 is calculated over a concatenated string (as raw data) of the following values: requestHttpMethod, requestUri, requestTimeStamp, nonce, requestContentBase64String

Online Payment HTML Gateway

[Not yet implemented]

This application provides online payment support for integration with external merchant websites via the browser. The main advantage that it provides to merchants is the fact that the merchant website does not need to collect payment data (such as credit card information)

The following diagram describes the intended flow:



1: Browser sends a HTTP GET request to a merchant website.

2: The response from merchant website returns some content to the browser including an HTML form with action = RoboBank HTML gateway URL and method = POST. The form should include some RoboBank -specific hidden fields among which website key, amount, currency, invoice number and a digital signature are mandatory.

3: Browser posts the HTML form rendered by merchant website to RoboBank HTML Gateway. At this point RoboBank system can identify the merchant customer (legal person) from the received website key that is mapped to customer Id in the data managed by the Merchant Service in the Middle tier. This website key can be configured by the customer in the Banking Portal (functionality available only for customers with legal person)

4: The response from RoboBank HTML gateway returns an HTML form for collecting card information (name, number, expiration and CVV)

5: Browser posts the HTML form back to RoboBank HTML gateway

The merchant website can also include some return url in the form rendered at step #2. This would enable RoboBank HTML gateway to redirect the browser back to merchant website after step #5.

The authentication should be implemented using a custom digital signature that should be included by merchant website in the form rendered at step #2 above in order to have the browser posting this to RoboBank HTML gateway. This signature is a hash of all parameters

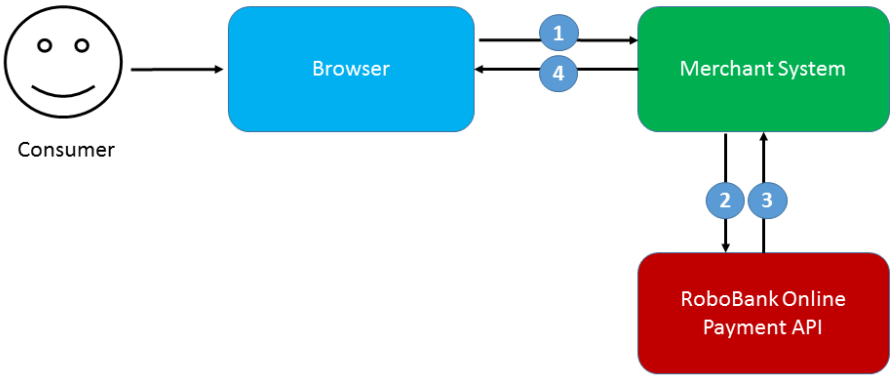
from the payment request using a secret key that a customer can configure in the Banking portal. (functionality available only for customers with legal person)

Online Payment API

[Not yet implemented]

This application provides online payment support for server-side integration with external merchant websites. The main advantage that it provides to merchants is the fact that they can customize the payment data UI according to their needs. (card name, number, expiration and CVV)

The following diagram describes the intended flow:



1: Browser sends a HTTP POST request to a merchant website including the card and amount information. (posting a form previously rendered by merchant website)

2: Merchant system sends a HTTP POST request including card and amount data to RoboBank Online Payment API.

3: RoboBank Online Payment API returns a response with the status of the transaction to merchant website.

4: Merchant website renders some content reflecting the transaction status to browser.

Authentication should be implemented as a custom HMAC approach similar to what Buckaroo JSON Gateway requires:

For every request the client is required to pass a custom format token in HTTP Authorization header. This token includes the “hmac” sequence, the website key, a Base64 hash using HMAC SHA256 signed with the secret key, a nonce (random sequence of characters that should differ from for each request) and a request timestamp. The HMAC SHA265 is calculated over a concatenated string (as raw data) of the following values: requestHttpMethod, requestUri, requestTimeStamp, nonce, requestContentBase64String

The website key and secret key can be configured by the customer in the Banking Portal (functionality available only for customers with legal person)

Sending the website key on every request allows RoboBank Online Payment API to identify the merchant customer (legal person) since the received website key is mapped to customer Id in the data managed by the Merchant Service in the Middle tier.