SNS Launch protocol v0.1

Overview

The social network standard (SNS) launch protocol is designed to integrate applications in portal like service.

Specifications

SNS Launch protocol technical specification

Test tooling

Use cases

UC Als gebruiker wil ik vanuit een wijkplatform drempelloos (zonder account aanmaken) naar een e-health module uit een e-health platform navigeren

Release notes

An overview of released versions

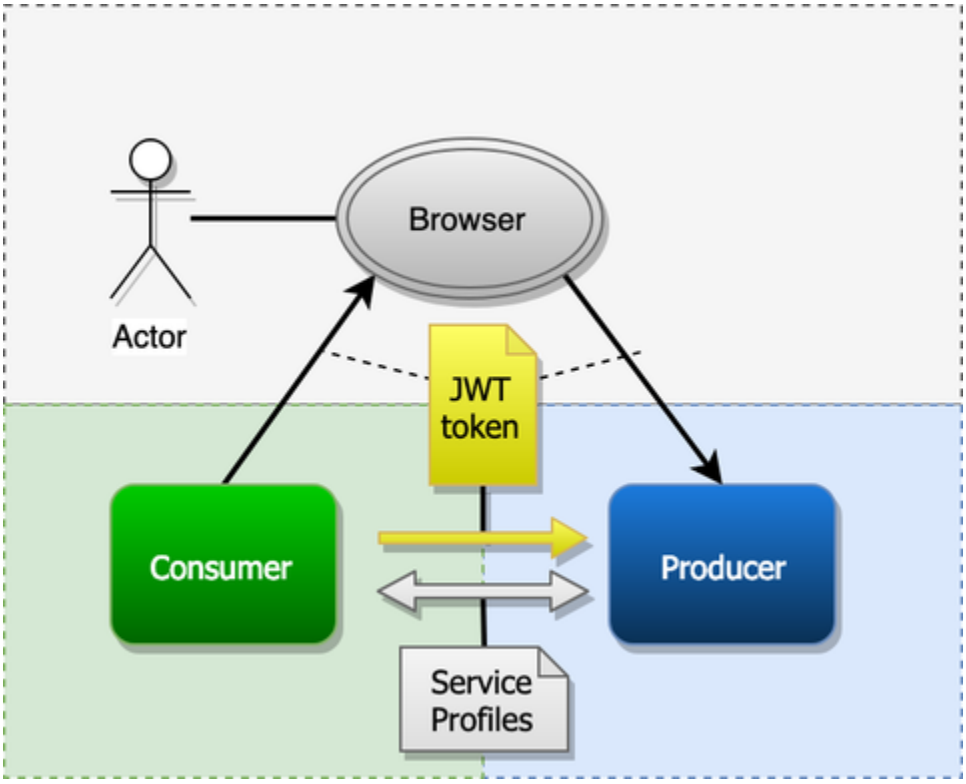
SNS Launch protocol technical specification

Architecture

The SNS launch protocol enables portal applications to integrate external applications like tools, games and treatments seamlessly into their platform. The SNS launch protocol connects applications like tools, games and treatment to a portal like environment. The portal, or consumer in this context, is the system that has an active session with an authenticated user in the system. The consumer prepares a launch by creating a JWT token that contains all the launch details needed for the producer to function properly. The producer in this context is the application that

delivers functionality to the user in the portal, either in the context of the portal as an iframe, or in its own context. The producer of the launch receives the JWT token and unpacks the information in the token to identify the user and the target treatment and launches a new session for the user.

As an extension to the basic version of the protocol as described above, the producer and consumer are able to communicate directly within the session of the user in order to exchange additional information or register progress and outcomes. The concept of these services are service *profil es*, each consumer and producer can implement and agree on the usage of various profiles that extend the basic usage.



Concepts

- **Consumer**, the portal like service that links to the producer, that is, an application like a tool, a game, or a treatment.
- **Producer**, the service that delivers an application like a tool, a game, or a treatment to the portal.
- **JWT token**, a package exchanged between consumer and producer that contains the relevant launch information.

Rationale

The SNS Launch protocol is highly inspired by the Learner Tool Interoperability (LTI) which has had a tremendous impact on the relation between learner management systems and tool providers. LTI has simplified the integration of external tools into learner management systems, the whole landscape of tool providers has emerged. The key concepts the LTI being successful has been:

- In the core the LTI standard is simple and clear.
- The LTI standard in its basic form is easy to integrate because it makes use of existing technologies and standards.
- The core standard can be extended by profiles; within LTI there are profiles for reading roster information and writing results.

The SNS launch standard applies these concepts when it comes to defining a successful launch protocol. The key differences are:

- Use of more modern technologies like JWT instead of OAuth 1.x.
- The alignment of user identity with a still to be specified SSO standard.
- More restrictions on security and the JWT validity.

The SNS Launch protocol has the following goals.

Ease of software implementation

- The protocol should be easy to implement, hours instead of days, and days instead of weeks. It does so by standing on the back of giants; that is make use of existing technologies and standards.

Ease of use and configuration

- The protocol should be easy to configure from both the producer and consumers' side. In the essence, an exchange of endpoint URL and public key pair should be sufficient.

Scalable, decentral, and point to point

- The architecture should not rely on external or central services and should be point-to-point in the sense that parties should be able to connect without relying on other parties and scale infinitely.

Secure

- The protocol should mitigate against most common attacks by aligning to pre-existing proven technologies like JWT.

Privacy

- The protocol should support anonymous identities and be reluctant to disseminate personal information.

Implementation guide

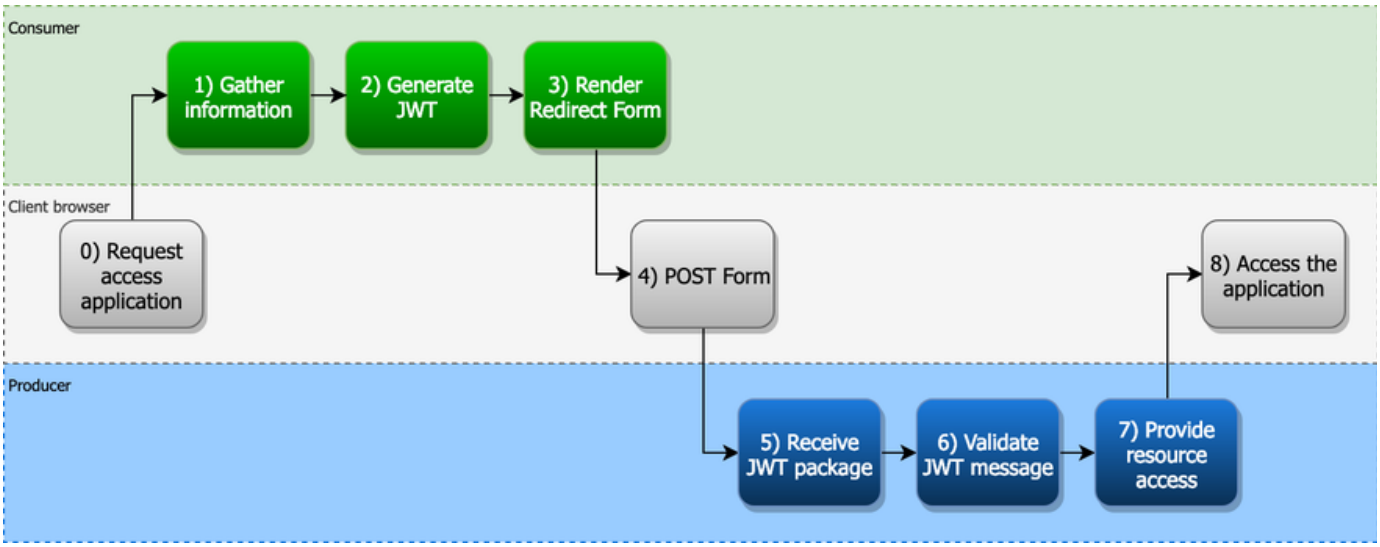
This guide describes how to implement the SNS Launch protocol. The protocol consists of:

- The communication protocol, how the interaction of the SNS Launch protocol looks like.
- The JWT message and the related payload.
- The SNS Launch protocol security restrictions.

Communication protocol

The core procedure of the launch looks as follows, the step

1. The client requests access to a (remote) application in the portal environment.
2. The consumer produces the information needed for the JWT, including:
    a. User identity
    b. Intended resource identifier.
    c. The JTW private and public key.
3. The consumer generates the JWT token based on the information above, described in more detail in the JWT message format section.
4. The consumer redirects the user to a html5 page with a form, which contains:
    a. The JWT token in a hidden field with name ***request***.
    b. The form method is ***post***.
    c. The action of the producers endpoint.
5. The client browser posts the redirect form (triggered by javascript).
6. The producer receives the post at the endpoint, and upacks the JWT token.
7. From the JWT token, it unpacks the fields and verifies the following:
    a. The identity of the producer (aud).
    b. The identity of the consumer (iss).
    c. Based on the identity of the producer, the signature of the issuer (iss).



The form-post-redirect message

In step 5) in the communication protocol, the user is redirected to the producer with the JWT message. The html5 sample below displays how this could be implemented.

```
<!doctype html>
<html>
<head>
<script>
window.onload = function () { document.forms[0].submit(); }
</script>
</head>
<body>
<form method="post" action="https://therapieland.nl/..." ...>
   <input type="hidden" name="request" value="<JWT Ticket>'" />
</form>
</body>
</html>
```

JWT message format

The message makes use of the JSON Web Token (JWT) standard. The standard is documented here: https://jwt.io/. Implementations in various languages are widely available. The concept of a JWT token is it consists of a header containing metadata of the token, a body or payload that consists of a set of required fields, and a signature that should be validated.

The JWT message consists out of the following fields, the fields with an asterix (*) are required.

Description	Field	Value
User identity*	sub	User unique identification, see format for details.
User email	email	User email
First name	given_name	User first name
Middle name	middle_name	User middle name
Last name	family_name	User last name
Subject*	resource_id	Identification of the target treatment
Producer*	iss	URL base of the producer
Domain*	aud	Base URL of the consumer.
Unique message id*	jti	UUID or anything else that makes the message unique
Issue time	iat	Timestamp from the time of creation
Expiration time*	exp	Timestamp for the expiration time
Public / Private key*	-	Signing private key, public key for validation.

User identifier format

The format for the user identity is an urn. This identifier is prefixed urn:sns:user, subsequently the reverse domain of the identity platform and finally the user identity. The format is as follows:

```
urn:sns:user:<domain>:<user>
```

For example:

```
urn:sns:user:nl.wikiwijk:123456
```

Security restrictions

- The JWT must use an async public / private key to sign the JWT tokens. The public key should be made available to the producer, the private key should remain private on the consumers infrastructure. The use of shared secrets is not allowed, because the issuer of the JWT cannot guarantee ownership as the key is shared.
    - All algorithms starting with HS should NOT be used, that is HS256, HS384, HS512
    - The following algorithms can be used by the consumer, the producer should support all algorithms:
        - RS256, recommended
        - RS384, optional
        - RS512, optional
        - ES256, recommended
        - ES384, optional
        - ES512, optional
- The expiration time (exp) on the message should be set to 5 minutes in order to prevent leaking JWT keys to be valid outside a timeframe.
- The unique message id (jti) should be verified as a nonce and should be based on a random or pseudo random number. If a UUID is used, it should be initialized with a random number. This approach mitigates replay attacks.
- Tokens must be transported over HTTPS from both consumer and producer sides.

Example message

```
{
  "alg": "RS256",
  "typ": "JWT"
}
{
  "sub": "urn:sns:user:nl.wikiwijk:123456",
  "aud": "therapieland.nl",
  "iss": "wikiwijk.nl",
  "resource_id": "paniek",
  "last_name": "Vries",
  "middle_name": "de",
  "exp": 1550663222,
  "iat": 1550662922,
  "first_name": "Klaas",
  "jti": "a5d155b2-d8b4-43bb-8730-1646ae35357c",
  "email": "klaas@devries.nl"
}
```

Launch configuration requirements

Producer configuration requirements

Field	Remark	Scope
Application URL	The endpoint of the producer application.	Per application
Public / Private key	The public / private keys	Preferably per application

Note that SamenBeter expects links to open in a new tab.

Consumer configuration requirements

Field	Remark	Scope
Consumer public key	The key to validate the consumer JWT message with.	Per consumer, based on the iss field value.

Appendix A, test keys and secrets

Test key and secret, please never use outside a test context.

Public key

Type: RSA

Length: 2024

```
MIIBHjANBgkqhkiG9w0BAQEFAAOCAQsAMIIBBgKB/gC+0zqjfI2zKvvjwUwE4JiLYyUqazpx
WD+hmyLCEXgzfbHIWvwRD54M8PJqCt+9Iq3PBIvpZoJezQ5rztEWN6OI7qoXq4ygZ4YTXGU+
ErfqLlvyMv/PfbuHU7oRS+4W0iq2mPwQQXSKMDJz4qSORa75p6xMMHd38xJgHQ6tBwPFMbwh
pGsGpCFpxRqlMR735D8gRbhFbSexxMhbyqpQTro0u6xPFoAecldiCJ8KNlp2/NNcRgMZKVIU
3rwhp52JcnI90by8UZoD0ItlRoXdaBmmQORWRrm2SC1rRu+KFidzjxe2cRiFVXqthqe1Ttm2
9atUeVftJhEgb7UpxKJPAgMBAAE=
```

Private Key

Type: RSA

Length: 2024

```
MIIEpwIBADANBgkqhkiG9w0BAQEFAASCBJEwggSNAgEAAoH+AL7TOqN8jbMq++PBTATgmItj
JSprOnFYP6GbIsIReDN9scha/BEPngzw8moK370irc8Ei+lmgl7NDmvO0RY3o4juqherjKBn
hhNcZT4St+ouW/Iy/899u4dTuhFL7hbSKraY/BBBdIowMnPipI5FrvmnrEwwd3fzEmAdDq0H
A8UxvCGkawakIWnFGqUxHvfkPyBFuEVtJ7HEyFvKqlBOujS7rE8WgB5yV2IInwo2Wnb801xG
AxkpUhTevCGnnYlycj3RvLxRmgPQi2VGhd1oGaZA5FZGubZILWtG74oWJ3OPF7ZxGIVVeq2G
p7VO2bb1q1R5V+0mESBvtSnEok8CAwEAAQKB/VO7cg6Mt8y3fsHIbqfxOV5oScWcOY/Erl8m
KJFJgxns/JayvcpqtOpuy6AWV2ixj9y33QC0V15r0fkiTgLWtS5/sykhwFoeMunJ8C7Vndfn
MbdMA42zWRcfeRTf4YAoBlALPwePASklzu2ktJotH4MyvNrNpY5/nT+JYIgx/LxhIwk/HxJ6
uVYiFpAINfAGfBphcgxzKWnV23WvRYtrIJc/XXLvSxK08tvoZfm4c4quf1i3LpTc+1mZmT+j
efZoXQcWUnEbCk5Q/8gvDigHMbdOlTqT4/iNj/03PmueWsljiyhbXDYOVGJCaGQpeNaFnhil
XPrYEBkAvXIOg6ECfw7l7td0wyPP0vCYFcbQEr3qng9vg2ISVas8gIOU/OeKNSJ9+wbKWcd0
DAztxGShuqDZjBXj+RSEL1XrABjDpk9RqpgkBx3NNXEbCBnYg3+LU8HCtUBWi5amaJi8JH28
39cVXjdZbPXBPmp5S93SKjmuoiBas8oKITh0yEwwdb8CfwzPAeg765BhD4AmwSzoQRy6Sfxf
6R0Z8Uo9a2mxBiGSKPvX7zQMG384208FvTlaW3UoOAhSN6HsfBwWT9pzRIaWAkFP8CWxRiRq
zg20FYzTweQZOnqje6YRYSocX64l22zhqV3Y3DdqevIiGpxDFqFM8QXeaAcchCvg6LpTl3EC
fwqlC1RynwM1eLhjUhvti5aazjilKrCl/QQOhJx/lXwyaeitLvEZH7C9H+cU8+AbFmfbSJZT
fyLDl7bB5B3NnUTLSyLNizAl8WtRLyaYZsx41m15G1xO+gm3+MA4nbIhg6YAJINTp+CoJFqb
NDPX+EeimUCYziErv7TA7GRTs60Cfws28F+KnzzBjtXQmNCd5eymOwNKYovFXBt5XWOjyE96
boHa1ahHdYfVm0c8KipeL7eLaEv42JbgvOXGr1IAHJ6OFxliSUxnQ5e9H/6ljzzHZ3s0j5wz
KZ8EloNNZoTOxqk1h5oQtveaNl1seMoaf2TpPhq6WXDoidz1Ri9l4zECfmzg4k6Jo2YpZVAm
1xQU5SPYDawH4DNlWeTMnqBEwfZap7wu79zJkZYdCaegzabb/FxFSu0+21djZbq4+PdtsxIq
mg8pObu2s7z+BqC0iM5z01deygAfgP4NRzmQqvECiDmjKWxXZlzQNPxnlu3MJZMrfDXTSzDe
IBph1YOIag==
```

Appendix B, near future roadmap

Near future developments will consists of the following

- Alignment with the still to be developed login and identity part of the SNS protocol, the impact probably will be that the JWT message will contain information about role. Another impact might be that the JWT message will get a higher exp date, matching something like a login session (30~60 minutes)
- Extension with profiles / services. The protocol will be extended to support communication between producer and consumer. This will be done by the consumer providing endpoints to the consumer at launch time.

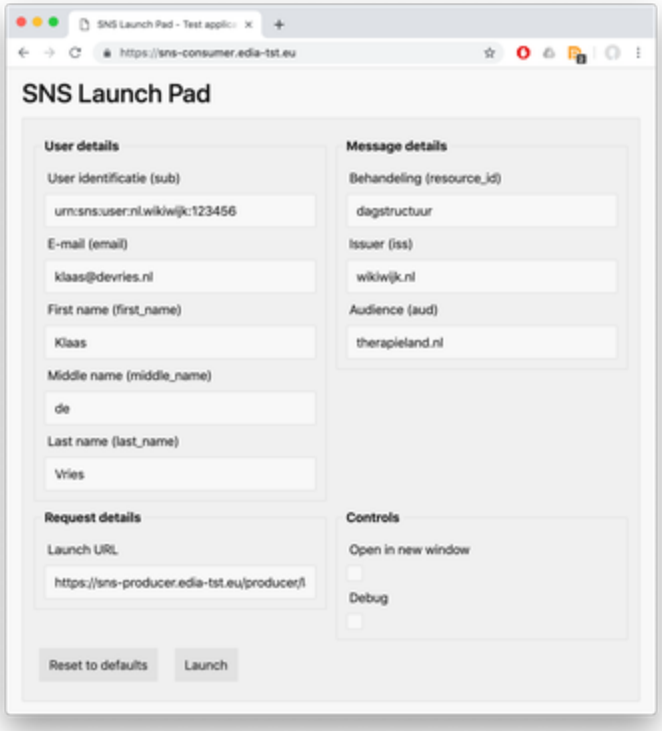
Appendix C, Test tools and validators

Producer test tool

The SNS launch producers can test with the following tool:

https://sns-consumer.edia-tst.eu/

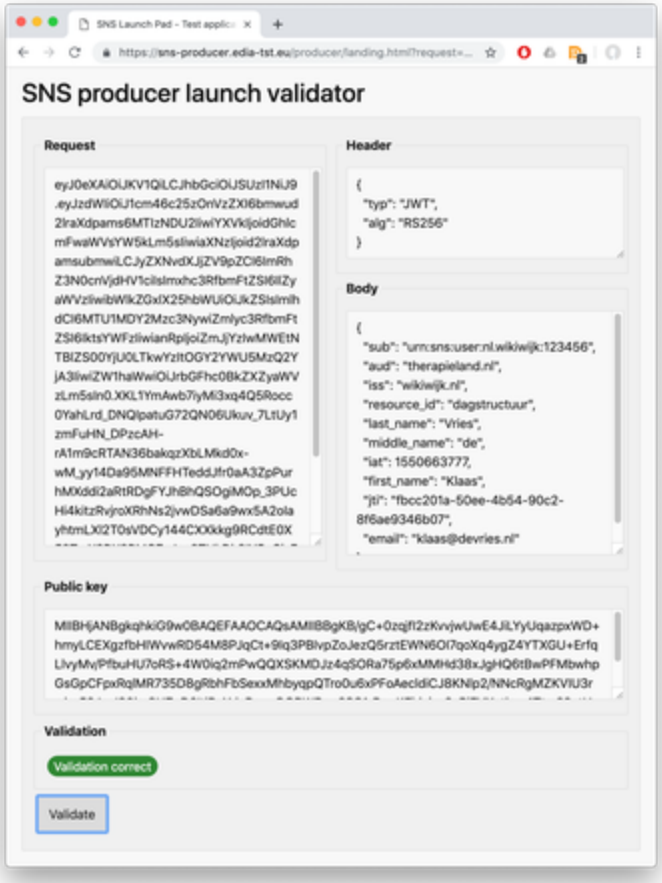
The tools allows to send a SNS Launch request to a given endpoint, and makes use of the test key and secret as provided in Appendix A.



Consumer test tool.

The tool consumer can use as endpoint the following:

https://sns-producer.edia-tst.eu/validate.html

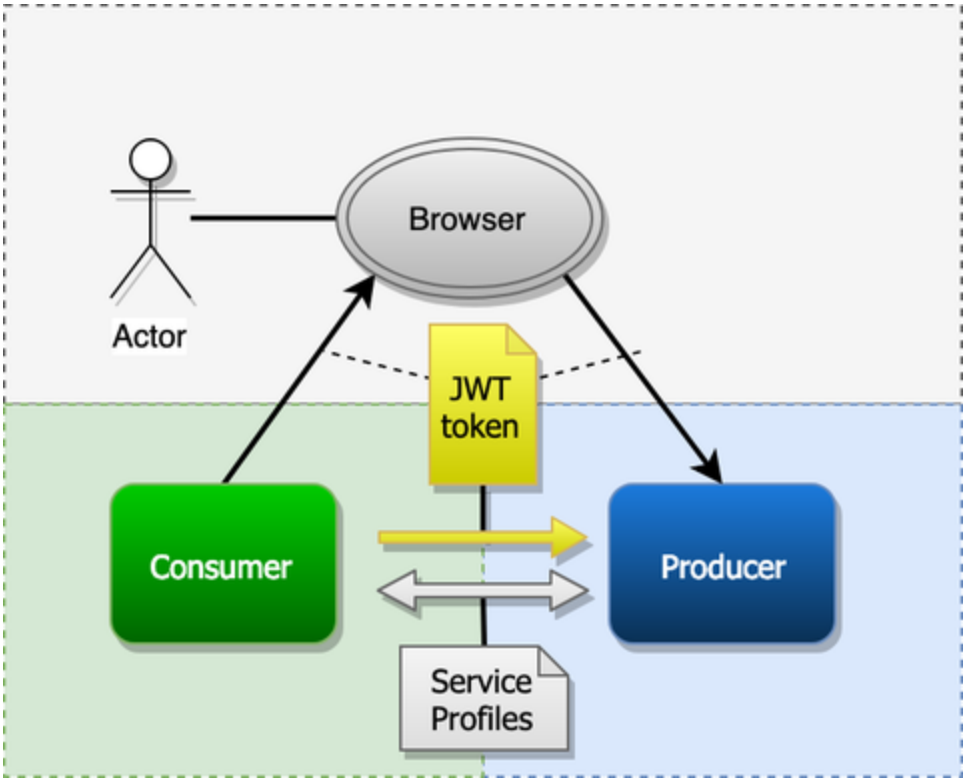


SNS Launch procedure architecture

Architecture

The SNS launch protocol enables portal applications to integrate external applications like tools, games and treatments seamlessly into their platform. The SNS launch protocol connects applications like tools, games and treatment to a portal like environment. The portal, or consumer in this context, is the system that has an active session with an authenticated user in the system. The consumer prepares a launch by creating a JWT token that contains all the launch details needed for the producer to function properly. The producer in this context is the application that delivers functionality to the user in the portal, either in the context of the portal as an iframe, or in its own context. The producer of the launch receives the JWT token and unpacks the information in the token to identify the user and the target treatment and launches a new session for the user.

As an extension to the basic version of the protocol as described above, the producer and consumer are able to communicate directly within the session of the user in order to exchange additional information or register progress and outcomes. The concept of these services are service *profil es*, each consumer and producer can implement and agree on the usage of various profiles that extend the basic usage.



Concepts

- **Consumer**, the portal like service that links to the producer, that is, an application like a tool, a game, or a treatment.
- **Producer**, the service that delivers an application like a tool, a game, or a treatment to the portal.
- **JWT token**, a package exchanged between consumer and producer that contains the relevant launch information.

Rationale

The SNS Launch protocol is highly inspired by the Learner Tool Interoperability (LTI) which has had a tremendous impact on the relation between learner management systems and tool providers. LTI has simplified the integration of external tools into learner management systems, the whole landscape of tool providers has emerged. The key concepts the LTI being successful has been:

- In the core the LTI standard is simple and clear.
- The LTI standard in its basic form is easy to integrate because it makes use of existing technologies and standards.
- The core standard can be extended by profiles; within LTI there are profiles for reading roster information and writing results.

The SNS launch standard applies these concepts when it comes to defining a successful launch protocol. The key differences are:

- Use of more modern technologies like JWT instead of OAuth 1.x.
- The alignment of user identity with a still to be specified SSO standard.
- More restrictions on security and the JWT validity.

The SNS Launch protocol has the following goals.

Ease of software implementation

- The protocol should be easy to implement, hours instead of days, and days instead of weeks. It does so by standing on the back of giants; that is make use of existing technologies and standards.

Ease of use and configuration

- The protocol should be easy to configure from both the producer and consumers' side. In the essence, an exchange of endpoint URL and public key pair should be sufficient.

Scalable, decentral, and point to point

- The architecture should not rely on external or central services and should be point-to-point in the sense that parties should be able to connect without relying on other parties and scale infinitely.

Secure

- The protocol should mitigate against most common attacks by aligning to pre-existing proven technologies like JWT.

Privacy

- The protocol should support anonymous identities and be reluctant to disseminate personal information.

SNS Launch protocol implementation guide

Implementation guide

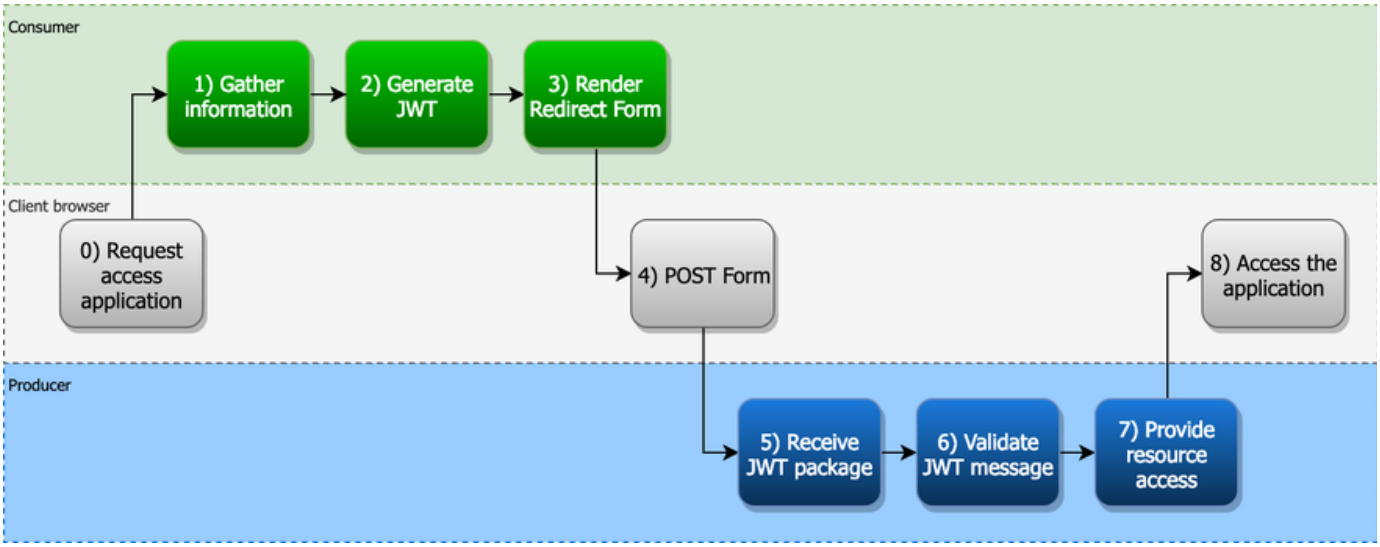
This guide describes how to implement the SNS Launch protocol. The protocol consists of:

- The communication protocol, how the interaction of the SNS Launch protocol looks like.
- The JWT message and the related payload.
- The SNS Launch protocol security restrictions.

Communication protocol

The core procedure of the launch looks as follows, the step

1. The client requests access to a (remote) application in the portal environment.
2. The consumer produces the information needed for the JWT, including:
    a. User identity
    b. Intended resource identifier.
    c. The JTW private and public key.
3. The consumer generates the JWT token based on the information above, described in more detail in the JWT message format section.
4. The consumer redirects the user to a html5 page with a form, which contains:
    a. The JWT token in a hidden field with name ***request***.
    b. The form method is ***post***.
    c. The action of the producers endpoint.
5. The client browser posts the redirect form (triggered by javascript).
6. The producer receives the post at the endpoint, and upacks the JWT token.
7. From the JWT token, it unpacks the fields and verifies the following:
    a. The identity of the producer (aud).
    b. The identity of the consumer (iss).
    c. Based on the identity of the producer, the signature of the issuer (iss).



THE FORM-POST-REDIRECT MESSAGE

In step 5) in the communication protocol, the user is redirected to the producer with the JWT message. The html5 sample below displays how this could be implemented.

```
<!doctype html>
<html>
<head>
<script>
window.onload = function () { document.forms[0].submit(); }
</script>
</head>
<body>
<form method="post" action="https://therapieland.nl/..." ...>
   <input type="hidden" name="request" value="<JWT Ticket>'" />
</form>
</body>
</html>
```

JWT message format

The message makes use of the JSON Web Token (JWT) standard. The standard is documented here: https://jwt.io/. Implementations in various languages are widely available. The concept of a JWT token is it consists of a header containing metadata of the token, a body or payload that consists of a set of required fields, and a signature that should be validated.

The JWT message consists out of the following fields, the fields with an asterix (*) are required.

Description	Field	Value
User identity*	sub	User unique identification, see format for details.
User email	email	User email
First name	given_name	User first name
Middle name	middle_name	User middle name
Last name	family_name	User last name
Subject*	resource_id	Identification of the target treatment
Producer*	iss	URL base of the producer
Domain*	aud	Base URL of the consumer.
Unique message id*	jti	UUID or anything else that makes the message unique
Issue time	iat	Timestamp from the time of creation
Expiration time*	exp	Timestamp for the expiration time
Public / Private key*	-	Signing private key, public key for validation.

USER IDENTIFIER FORMAT

The format for the user identity is an urn. This identifier is prefixed urn:sns:user, subsequently the reverse domain of the identity platform and finally the user identity. The format is as follows:

```
urn:sns:user:<domain>:<user>
```

For example:

```
urn:sns:user:nl.wikiwijk:123456
```

Security restrictions

- The JWT must use an async public / private key to sign the JWT tokens. The public key should be made available to the producer, the private key should remain private on the consumers infrastructure. The use of shared secrets is not allowed, because the issuer of the JWT cannot guarantee ownership as the key is shared.
    - All algorithms starting with HS should NOT be used, that is HS256, HS384, HS512
    - The following algorithms can be used by the consumer, the producer should support all algorithms:
        - RS256, recommended
        - RS384, optional
        - RS512, optional
        - ES256, recommended
        - ES384, optional
        - ES512, optional
- The expiration time (exp) on the message should be set to 5 minutes in order to prevent leaking JWT keys to be valid outside a timeframe.
- The unique message id (jti) should be verified as a nonce and should be based on a random or pseudo random number. If a UUID is used, it should be initialized with a random number. This approach mitigates replay attacks.
- Tokens must be transported over HTTPS from both consumer and producer sides.

EXAMPLE MESSAGE

```
{
  "alg": "RS256",
  "typ": "JWT"
}
{
  "sub": "urn:sns:user:nl.wikiwijk:123456",
  "aud": "therapieland.nl",
  "iss": "wikiwijk.nl",
  "resource_id": "paniek",
  "last_name": "Vries",
  "middle_name": "de",
  "exp": 1550663222,
  "iat": 1550662922,
  "first_name": "Klaas",
  "jti": "a5d155b2-d8b4-43bb-8730-1646ae35357c",
  "email": "klaas@devries.nl"
}
```

Launch configuration requirements

PRODUCER CONFIGURATION REQUIREMENTS

Field	Remark	Scope
Application URL	The endpoint of the producer application.	Per application
Public / Private key	The public / private keys	Preferably per application

NOTE THAT SAMENBETER EXPECTS LINKS TO OPEN IN A NEW TAB.

CONSUMER CONFIGURATION REQUIREMENTS

Field	Remark	Scope
Consumer public key	The key to validate the consumer JWT message with.	Per consumer, based on the iss field value.

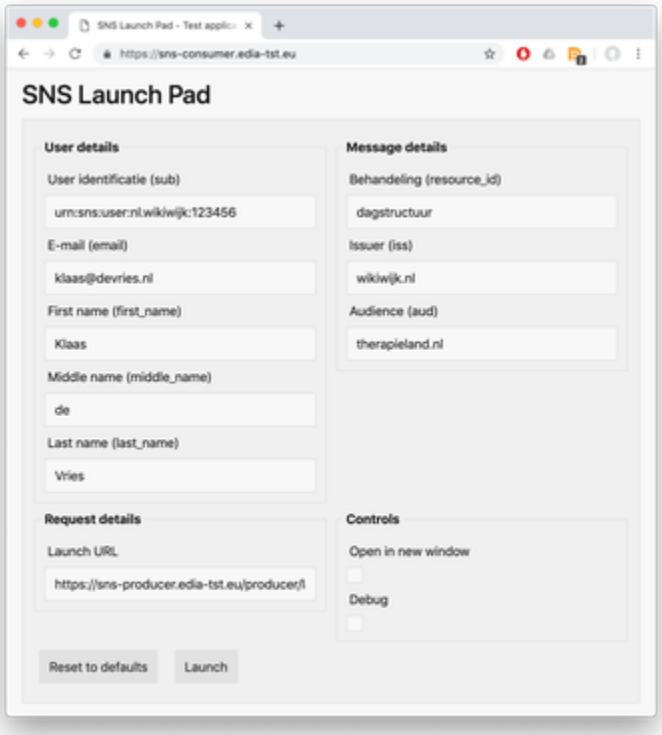
Appendix A, SNS Launch protocol test tools and validators

Producer test tool

The SNS launch producers can test with the following tool:

https://sns-consumer.edia-tst.eu/

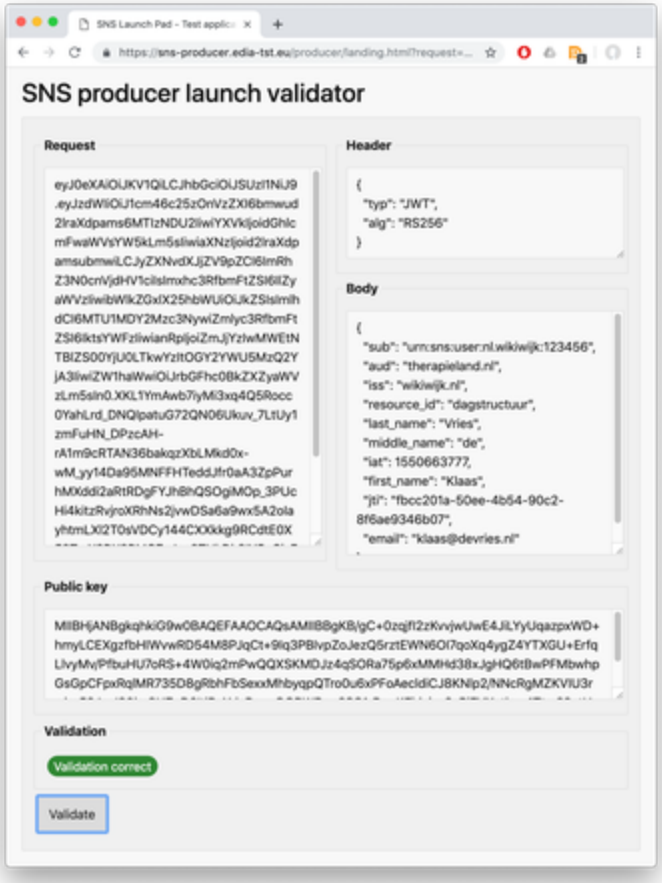
The tools allows to send a SNS Launch request to a given endpoint, and makes use of the test key and secret as provided in Appendix A.



Consumer test tool.

The tool consumer can use as endpoint the following:

https://sns-producer.edia-tst.eu/validate.html



Appendix B, near future roadmap

Near future developments will consists of the following

- Alignment with the still to be developed login and identity part of the SNS protocol, the impact probably will be that the JWT message will contain information about role. Another impact might be that the JWT message will get a higher exp date, matching something like a login session (30~60 minutes)
- Extension with profiles / services. The protocol will be extended to support communication between producer and consumer. This will be done by the consumer providing endpoints to the consumer at launch time.

Appendix C, test keys and secrets

Test key and secret, please never use outside a test context.

Public key

Type: RSA

Length: 2024

```
MIIBHjANBgkqhkiG9w0BAQEFAAOCAQsAMIIBBgKB/gC+0zqjfI2zKvvjwUwE4JiLYyUqazpx
WD+hmyLCEXgzfbHIWvwRD54M8PJqCt+9Iq3PBIvpZoJezQ5rztEWN6OI7qoXq4ygZ4YTXGU+
ErfqLlvyMv/PfbuHU7oRS+4W0iq2mPwQQXSKMDJz4qSORa75p6xMMHd38xJgHQ6tBwPFMbwh
pGsGpCFpxRqlMR735D8gRbhFbSexxMhbyqpQTro0u6xPFoAecldiCJ8KNlp2/NNcRgMZKVIU
3rwhp52JcnI90by8UZoD0ItlRoXdaBmmQORWRrm2SC1rRu+KFidzjxe2cRiFVXqthqe1Ttm2
9atUeVftJhEgb7UpxKJPAgMBAAE=
```

Private Key

Type: RSA

Length: 2024

```
MIIEpwIBADANBgkqhkiG9w0BAQEFAASCBJEwggSNAgEAAoH+AL7TOqN8jbMq++PBTATgmItj
JSprOnFYP6GbIsIReDN9scha/BEPngzw8moK370irc8Ei+lmgl7NDmvO0RY3o4juqherjKBn
hhNcZT4St+ouW/Iy/899u4dTuhFL7hbSKraY/BBBdIowMnPipI5FrvmnrEwwd3fzEmAdDq0H
A8UxvCGkawakIWnFGqUxHvfkPyBFuEVtJ7HEyFvKqlBOujS7rE8WgB5yV2IInwo2Wnb801xG
AxkpUhTevCGnnYlycj3RvLxRmgPQi2VGhd1oGaZA5FZGubZILWtG74oWJ3OPF7ZxGIVVeq2G
p7VO2bb1q1R5V+0mESBvtSnEok8CAwEAAQKB/VO7cg6Mt8y3fsHIbqfxOV5oScWcOY/Erl8m
KJFJgxns/JayvcpqtOpuy6AWV2ixj9y33QC0V15r0fkiTgLWtS5/sykhwFoeMunJ8C7Vndfn
MbdMA42zWRcfeRTf4YAoBlALPwePASklzu2ktJotH4MyvNrNpY5/nT+JYIgx/LxhIwk/HxJ6
uVYiFpAINfAGfBphcgxzKWnV23WvRYtrIJc/XXLvSxK08tvoZfm4c4quf1i3LpTc+1mZmT+j
efZoXQcWUnEbCk5Q/8gvDigHMbdOlTqT4/iNj/03PmueWsljiyhbXDYOVGJCaGQpeNaFnhil
XPrYEBkAvXIOg6ECfw7l7td0wyPP0vCYFcbQEr3qng9vg2ISVas8gIOU/OeKNSJ9+wbKWcd0
DAztxGShuqDZjBXj+RSEL1XrABjDpk9RqpgkBx3NNXEbCBnYg3+LU8HCtUBWi5amaJi8JH28
39cVXjdZbPXBPmp5S93SKjmuoiBas8oKITh0yEwwdb8CfwzPAeg765BhD4AmwSzoQRy6Sfxf
6R0Z8Uo9a2mxBiGSKPvX7zQMG384208FvTlaW3UoOAhSN6HsfBwWT9pzRIaWAkFP8CWxRiRq
zg20FYzTweQZOnqje6YRYSocX64l22zhqV3Y3DdqevIiGpxDFqFM8QXeaAcchCvg6LpTl3EC
fwqlC1RynwM1eLhjUhvti5aazjilKrCl/QQOhJx/lXwyaeitLvEZH7C9H+cU8+AbFmfbSJZT
fyLDl7bB5B3NnUTLSyLNizAl8WtRLyaYZsx41m15G1xO+gm3+MA4nbIhg6YAJINTp+CoJFqb
NDPX+EeimUCYziErv7TA7GRTs60Cfws28F+KnzzBjtXQmNCd5eymOwNKYovFXBt5XWOjyE96
boHa1ahHdYfVm0c8KipeL7eLaEv42JbgvOXGr1IAHJ6OFxliSUxnQ5e9H/6ljzzHZ3s0j5wz
KZ8EloNNZoTOxqk1h5oQtveaNl1seMoaf2TpPhq6WXDoidz1Ri9l4zECfmzg4k6Jo2YpZVAm
1xQU5SPYDawH4DNlWeTMnqBEwfZap7wu79zJkZYdCaegzabb/FxFSu0+21djZbq4+PdtsxIq
mg8pObu2s7z+BqC0iM5z01deygAfgP4NRzmQqvECiDmjKWxXZlzQNPxnlu3MJZMrfDXTSzDe
IBph1YOIag==
```

SNS Protocol code examples

Java examples

Example 1, generate a SNS Launch token with an RSA key

This example makes use of the auth0 JWT library. The key algorithm used is RSA.

```
import com.auth0.jwt.JWT;
import com.auth0.jwt.algorithms.Algorithm;
```

```
import org.apache.commons.codec.binary.Base64;

import java.security.KeyFactory;
import java.security.interfaces.RSAPrivateKey;
import java.security.spec.PKCS8EncodedKeySpec;
import java.util.Date;
import java.util.UUID;

public class JwtConsumerExample {
 public static void main(String[] args) throws Exception {
  String resourceId = "dagstructuur";
  String subject = "urn:sns:user:wikiwijk.nl:123456";
  String issuer = "wikiwijk.nl";
  String audience = "therapieland.nl";
  String email = "klaas@devries.nl";
  String firstName = "Klaas";
  String middleName = "de";
  String lastName = "Vries";
  String privateK =
"MIIEpwIBADANBgkqhkiG9w0BAQEFAASCBJEwggSNAgEAAoH+AL7TOqN8jbMq++PBTATgmIt
jJSprOnFYP6GbIsIReDN9scha/BEPngzw8moK370irc8Ei+lmgl7NDmvO0RY3o4juqherjKB
nhhNcZT4St+ouW/Iy/899u4dTuhFL7hbSKraY/BBBdIowMnPipI5FrvmnrEwwd3fzEmAdDq0
HA8UxvCGkawakIWnFGqUxHvfkPyBFuEVtJ7HEyFvKqlBOujS7rE8WgB5yV2IInwo2Wnb801x
GAxkpUhTevCGnnYlycj3RvLxRmgPQi2VGhd1oGaZA5FZGubZILWtG74oWJ3OPF7ZxGIVVeq2
Gp7VO2bb1q1R5V+0mESBvtSnEok8CAwEAAQKB/VO7cg6Mt8y3fsHIbqfxOV5oScWcOY/Erl8
mKJFJgxns/JayvcpqtOpuy6AWV2ixj9y33QC0V15r0fkiTgLWtS5/sykhwFoeMunJ8C7Vndf
nMbdMA42zWRcfeRTf4YAoBlALPwePASklzu2ktJotH4MyvNrNpY5/nT+JYIgx/LxhIwk/HxJ
6uVYiFpAINfAGfBphcgxzKWnV23WvRYtrIJc/XXLvSxK08tvoZfm4c4quf1i3LpTc+1mZmT+
jefZoXQcWUnEbCk5Q/8gvDigHMbdOlTqT4/iNj/03PmueWsljiyhbXDYOVGJCaGQpeNaFnhi
lXPrYEBkAvXIOg6ECfw7l7td0wyPP0vCYFcbQEr3qng9vg2ISVas8gIOU/OeKNSJ9+wbKWcd
0DAztxGShuqDZjBXj+RSEL1XrABjDpk9RqpgkBx3NNXEbCBnYg3+LU8HCtUBWi5amaJi8JH2
839cVXjdZbPXBPmp5S93SKjmuoiBas8oKITh0yEwwdb8CfwzPAeg765BhD4AmwSzoQRy6Sfx
f6R0Z8Uo9a2mxBiGSKPvX7zQMG384208FvTlaW3UoOAhSN6HsfBwWT9pzRIaWAkFP8CWxRiR
qzg20FYzTweQZOnqje6YRYSocX64l22zhqV3Y3DdqevIiGpxDFqFM8QXeaAcchCvg6LpTl3E
CfwqlC1RynwM1eLhjUhvti5aazjilKrCl/QQOhJx/lXwyaeitLvEZH7C9H+cU8+AbFmfbSJZ
TfyLDl7bB5B3NnUTLSyLNizAl8WtRLyaYZsx41m15G1xO+gm3+MA4nbIhg6YAJINTp+CoJFq
bNDPX+EeimUCYziErv7TA7GRTs60Cfws28F+KnzzBjtXQmNCd5eymOwNKYovFXBt5XWOjyE9
6boHa1ahHdYfVm0c8KipeL7eLaEv42JbgvOXGr1IAHJ6OFxliSUxnQ5e9H/6ljzzHZ3s0j5w
zKZ8EloNNZoTOxqk1h5oQtveaNl1seMoaf2TpPhq6WXDoidz1Ri9l4zECfmzg4k6Jo2YpZVA
m1xQU5SPYDawH4DNlWeTMnqBEwfZap7wu79zJkZYdCaegzabb/FxFSu0+21djZbq4+PdtsxI
qmg8pObu2s7z+BqC0iM5z01deygAfgP4NRzmQqvECiDmjKWxXZlzQNPxnlu3MJZMrfDXTSzD
eIBph1YOIag==";
  KeyFactory keyFactory = KeyFactory.getInstance("RSA");
  RSAPrivateKey privateKey = (RSAPrivateKey) keyFactory.generatePrivate(
    new PKCS8EncodedKeySpec(Base64.decodeBase64(privateK)));


  String jwt = JWT.create()
    .withIssuedAt(new Date())
    .withJWTId(UUID.randomUUID().toString())
```

```
        .withSubject(subject)
        .withIssuer(issuer)
        .withAudience(audience)
        .withClaim("resource_id", resourceId)
        .withClaim("email", email)
        .withClaim("first_name", firstName)
        .withClaim("middle_name", middleName)
        .withClaim("last_name", lastName)
        .withExpiresAt(new Date(System.currentTimeMillis()+5*60*1000))
        .sign(Algorithm.RSA256(null, privateKey));
```

```
  System.out.println(jwt);
 }
}
```

Example 2: Validate a SNS Launch message

This example is more complicated, mostly because the auth0 JWT library has no helper method for selecting the right algorithm from the JWT header.

```
import com.auth0.jwt.JWT;
import com.auth0.jwt.algorithms.Algorithm;
import com.auth0.jwt.interfaces.DecodedJWT;
import org.apache.commons.codec.binary.Base64;

import java.security.KeyFactory;
import java.security.NoSuchAlgorithmException;
import java.security.interfaces.ECPublicKey;
import java.security.interfaces.RSAPublicKey;
import java.security.spec.InvalidKeySpecException;
import java.security.spec.X509EncodedKeySpec;

public class JwtProviderExample {
 public static void main(String[] args) throws Exception {
  String token = args[0];

  // Get the algorithm name from the JWT.
  String algorithmName = JWT.decode(token).getAlgorithm();
  // Get the issuer name from the JWT.
  String issuer = JWT.decode(token).getIssuer();

  // Lookup the issuer.
  String publicK = getPublicKeyForIssuer(issuer); // Public key from
appendix A

  // Get the algorithm from the public key and algorithm name.
  Algorithm algorithm = getAlgorithm(publicK, algorithmName);

  // Decode and verify the token.
  DecodedJWT jwt = JWT.require(algorithm)
    .withAudience("therapieland.nl") // Make sure to require yourself to
be the audience.
    .build()
    .verify(token);

  // Read the parameters from the jwt token.
  String subject = jwt.getSubject();
  String resourceId = jwt.getClaim("resource_id").asString();
  String email = jwt.getClaim("email").asString();
  String firstName = jwt.getClaim("first_name").asString();
```

```
  String middleName = jwt.getClaim("middle_name").asString();
  String lastName = jwt.getClaim("last_name").asString();

  System.out.println(String.format("The SNS launch recieved the user
with id %s for resource %s, the user email is %s, the user is known as
%s %s %s.",
    subject,
    resourceId,
    email,
    firstName,
    middleName,
    lastName));
 }

 /**
  * This method should lookup the public key configured with the issuer
from the configuration
  * and / or persistent storage.
  *
  * @param issuer the issuer from the JWT token.
  * @return a public key encoded as String
  */
 private static String getPublicKeyForIssuer(String issuer) {
  // Return the test key from Appendix A.
  return "..." ;
 }

 /**
  * Unfortunately, this implementation of JWT has no helper method for
selecting the right
  * algorithm from the header. The public key must match the algorithm
type (RSA or EC), but
  * the size of the hash algorithm can vary.
  *
  * @param publicKey
  * @param algorithmName
  * @return in instance of the {@link Algorithm} class.
  * @throws NoSuchAlgorithmException
  * @throws InvalidKeySpecException
  * @throws IllegalArgumentException if the algorithmName is not one of
RS{256,384,512} or ES{256,384,512}
  */
 private static Algorithm getAlgorithm(String publicKey, String
algorithmName) throws NoSuchAlgorithmException, InvalidKeySpecException,
IllegalArgumentException {
  switch (algorithmName) {
   case "RS256": {
    return Algorithm.RSA256(getRsaPublicKey(publicKey), null);
   }
   case "RS384": {
```

```
    return Algorithm.RSA384(getRsaPublicKey(publicKey), null);
   }
   case "RS512": {
    return Algorithm.RSA512(getRsaPublicKey(publicKey), null);
   }
   case "ES256": {
    return Algorithm.ECDSA256(getEcPublicKey(publicKey), null);
   }
   case "ES384": {
    return Algorithm.ECDSA384(getEcPublicKey(publicKey), null);
   }
   case "ES512": {
    return Algorithm.ECDSA512(getEcPublicKey(publicKey), null);
   }
   default:
    throw new IllegalArgumentException(String.format("Unsupported
algorithm %s", algorithmName));
  }

 }

 /**
  * Parses a public key to an instance of {@link ECPublicKey}.
  *
  * @param publicKey the string representation of the public key.
  * @return an instance of {@link ECPublicKey}.
  * @throws NoSuchAlgorithmException
  * @throws InvalidKeySpecException
  */
 private static ECPublicKey getEcPublicKey(String publicKey) throws
NoSuchAlgorithmException, InvalidKeySpecException {
  KeyFactory keyFactory = KeyFactory.getInstance("EC");
  return (ECPublicKey) keyFactory.generatePublic(
    new X509EncodedKeySpec(Base64.decodeBase64(publicKey)));
 }

 /**
  * Parses a public key to an instance of {@link RSAPublicKey}.
  *
  * @param publicKey the string representation of the public key.
  * @return an instance of {@link RSAPublicKey}.
  * @throws NoSuchAlgorithmException
  * @throws InvalidKeySpecException
  */
 private static RSAPublicKey getRsaPublicKey(String publicKey) throws
NoSuchAlgorithmException, InvalidKeySpecException {
  KeyFactory keyFactory = KeyFactory.getInstance("RSA");
  return (RSAPublicKey) keyFactory.generatePublic(
    new X509EncodedKeySpec(Base64.decodeBase64(publicKey)));
 }
```

```
}
```

Example 3: Generate a RSA key pair

```
import java.security.*;
import static org.apache.commons.codec.binary.Base64.encodeBase64String;

public class RsaKeyPairGenerator {

 public static void main(String[] args) throws Exception {
  new RsaKeyPairGenerator().generate();
 }

 public void generate() throws NoSuchAlgorithmException {
  // Create a new generator
  KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA");
  // Set the key size
  generator.initialize(2024);
  // Generate a pair
  KeyPair keyPair = generator.generateKeyPair();
  // Output the public key as base64
  String publicK = encodeBase64String(keyPair.getPublic().getEncoded());
  // Output the private key as base64
  String privateK =
encodeBase64String(keyPair.getPrivate().getEncoded());

  System.out.println(publicK);
  System.out.println(privateK);
 }
}
```

Example 4: Generate a EC key pair

```
import java.security.*;
import static org.apache.commons.codec.binary.Base64.encodeBase64String;

public class EcKeyPairGenerator {

 public static void main(String[] args) throws Exception {
  new EcKeyPairGenerator().generate();
 }

 public void generate() throws NoSuchAlgorithmException {
  // Create a new generator
  KeyPairGenerator generator = KeyPairGenerator.getInstance("EC");
  SecureRandom random = SecureRandom.getInstance("SHA1PRNG");
  // Set the key size and random
  generator.initialize(256, random);
  // Generate a pair
  KeyPair keyPair = generator.generateKeyPair();
  // Output the public key as base64
  String publicK = encodeBase64String(keyPair.getPublic().getEncoded());
  // Output the private key as base64
  String privateK =
encodeBase64String(keyPair.getPrivate().getEncoded());

  System.out.println(publicK);
  System.out.println(privateK);
 }
}
```

Python examples

Example 1, generate a SNS Launch token with an RSA key

```
import jwt
import time
from uuid import uuid4


def main():
    # The public key as provided by appendix A.
    private_key = '...'
    # Format as PEM key
    public_key_formatted = f'-----BEGIN PRIVATE KEY-----\n' \
        f'{private_key}' \
        f'\n-----END PRIVATE KEY-----'

    # Time function
    payload = {}
    payload['sub'] = 'urn:sns:user:wikiwijk.nl:123456'
    payload['aud'] = 'therapieland.nl'
    payload['iss'] = 'wikiwijk.nl'
    payload['resource_id'] = 'dagstructuur'
    payload['first_name'] = 'Klaas'
    payload['middle_name'] = 'de'
    payload['last_name'] = 'Vries'
    payload['email'] = 'klaas@devries.nl'
    payload['iat'] = time.time()
    payload['exp'] = time.time() + (5 * 60 * 1000)
    payload['jti'] = str(uuid4())

    jwt_encode = jwt.encode(payload, public_key_formatted,
algorithm='RS256').decode('utf8')
    print(jwt_encode)


if __name__ == '__main__':
    main()
```

Example 2: Validate a SNS Launch message

```
import sys
import jwt


def main(jwt_token):
    # The public key as provided by appendix A.
    public_key = '...'
    # Format as PEM key
    public_key_formatted = f'-----BEGIN PUBLIC KEY-----\n' \
        f'{public_key}' \
        f'\n-----END PUBLIC KEY-----'
    # Use the JWT decode, make sure to set the audience
    jwt_decode = jwt.decode(jwt_token, public_key_formatted,
                            audience="therapieland.nl")
    user_id = jwt_decode['sub']
    email = jwt_decode['email']
    first_name = jwt_decode['first_name']
    middle_name = jwt_decode['middle_name']
    last_name = jwt_decode['last_name']
    issuer = jwt_decode['iss']
    unique_message_id = jwt_decode['jti']
    treatment_id = jwt_decode['resource_id']
    print(f'User {first_name} {middle_name} {last_name} with email
{email} '
          f'from {issuer} wants to launch treatment {treatment_id} '
          f'with launch id {unique_message_id}')
```

Example 3: Generate a RSA key pair

```
from cryptography.hazmat.primitives import serialization as
crypto_serialization
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.backends import default_backend as
crypto_default_backend


def main():
    key = rsa.generate_private_key(
        backend=crypto_default_backend(),
        public_exponent=65537,
        key_size=2024
    )
    private_key = key.private_bytes(
        crypto_serialization.Encoding.PEM,
        crypto_serialization.PrivateFormat.PKCS8,
        crypto_serialization.NoEncryption())
    public_key = key.public_key().public_bytes(
        crypto_serialization.Encoding.PEM,
        crypto_serialization.PublicFormat.SubjectPublicKeyInfo
    )
    print('Public key {}'.format(public_key))
    print('Private key {}'.format(private_key))


if __name__ == '__main__':
    main()
```

Example 4: Generate a EC key pair

```
from cryptography.hazmat.backends import default_backend as
crypto_default_backend
from cryptography.hazmat.primitives import serialization as
crypto_serialization
from cryptography.hazmat.primitives.asymmetric import ec


def main():
    key = ec.generate_private_key(
        curve=ec.SECP256K1,
        backend=crypto_default_backend()
    )
    private_key = key.private_bytes(
        crypto_serialization.Encoding.PEM,
        crypto_serialization.PrivateFormat.PKCS8,
        crypto_serialization.NoEncryption())
    public_key = key.public_key().public_bytes(
        crypto_serialization.Encoding.PEM,
        crypto_serialization.PublicFormat.SubjectPublicKeyInfo
    )
    print('Public key {}'.format(public_key))
    print('Private key {}'.format(private_key))


if __name__ == '__main__':
    main()
```

SNS Launch protocol Java examples

Example 1, generate a SNS Launch token with an RSA key

This example makes use of the auth0 JWT library. The key algorithm used is RSA.

```
import com.auth0.jwt.JWT;
import com.auth0.jwt.algorithms.Algorithm;
import org.apache.commons.codec.binary.Base64;

import java.security.KeyFactory;
import java.security.interfaces.RSAPrivateKey;
import java.security.spec.PKCS8EncodedKeySpec;
import java.util.Date;
import java.util.UUID;

public class JwtConsumerExample {
 public static void main(String[] args) throws Exception {
  String resourceId = "dagstructuur";
```

```
  String subject = "urn:sns:user:wikiwijk.nl:123456";
  String issuer = "wikiwijk.nl";
  String audience = "therapieland.nl";
  String email = "klaas@devries.nl";
  String firstName = "Klaas";
  String middleName = "de";
  String lastName = "Vries";
  String privateK =
"MIIEpwIBADANBgkqhkiG9w0BAQEFAASCBJEwggSNAgEAAoH+AL7TOqN8jbMq++PBTATgmIt
jJSprOnFYP6GbIsIReDN9scha/BEPngzw8moK370irc8Ei+lmgl7NDmvO0RY3o4juqherjKB
nhhNcZT4St+ouW/Iy/899u4dTuhFL7hbSKraY/BBBdIowMnPipI5FrvmnrEwwd3fzEmAdDq0
HA8UxvCGkawakIWnFGqUxHvfkPyBFuEVtJ7HEyFvKqlBOujS7rE8WgB5yV2IInwo2Wnb801x
GAxkpUhTevCGnnYlycj3RvLxRmgPQi2VGhd1oGaZA5FZGubZILWtG74oWJ3OPF7ZxGIVVeq2
Gp7VO2bb1q1R5V+0mESBvtSnEok8CAwEAAQKB/VO7cg6Mt8y3fsHIbqfxOV5oScWcOY/Erl8
mKJFJgxns/JayvcpqtOpuy6AWV2ixj9y33QC0V15r0fkiTgLWtS5/sykhwFoeMunJ8C7Vndf
nMbdMA42zWRcfeRTf4YAoBlALPwePASklzu2ktJotH4MyvNrNpY5/nT+JYIgx/LxhIwk/HxJ
6uVYiFpAINfAGfBphcgxzKWnV23WvRYtrIJc/XXLvSxK08tvoZfm4c4quf1i3LpTc+1mZmT+
jefZoXQcWUnEbCk5Q/8gvDigHMbdOlTqT4/iNj/03PmueWsljiyhbXDYOVGJCaGQpeNaFnhi
lXPrYEBkAvXIOg6ECfw7l7td0wyPP0vCYFcbQEr3qng9vg2ISVas8gIOU/OeKNSJ9+wbKWcd
0DAztxGShuqDZjBXj+RSEL1XrABjDpk9RqpgkBx3NNXEbCBnYg3+LU8HCtUBWi5amaJi8JH2
839cVXjdZbPXBPmp5S93SKjmuoiBas8oKITh0yEwwdb8CfwzPAeg765BhD4AmwSzoQRy6Sfx
f6R0Z8Uo9a2mxBiGSKPvX7zQMG384208FvTlaW3UoOAhSN6HsfBwWT9pzRIaWAkFP8CWxRiR
qzg20FYzTweQZOnqje6YRYSocX64l22zhqV3Y3DdqevIiGpxDFqFM8QXeaAcchCvg6LpTl3E
CfwqlC1RynwM1eLhjUhvti5aazjilKrCl/QQOhJx/lXwyaeitLvEZH7C9H+cU8+AbFmfbSJZ
TfyLDl7bB5B3NnUTLSyLNizAl8WtRLyaYZsx41m15G1xO+gm3+MA4nbIhg6YAJINTp+CoJFq
bNDPX+EeimUCYziErv7TA7GRTs60Cfws28F+KnzzBjtXQmNCd5eymOwNKYovFXBt5XWOjyE9
6boHa1ahHdYfVm0c8KipeL7eLaEv42JbgvOXGr1IAHJ6OFxliSUxnQ5e9H/6ljzzHZ3s0j5w
zKZ8EloNNZoTOxqk1h5oQtveaNl1seMoaf2TpPhq6WXDoidz1Ri9l4zECfmzg4k6Jo2YpZVA
m1xQU5SPYDawH4DNlWeTMnqBEwfZap7wu79zJkZYdCaegzabb/FxFSu0+21djZbq4+PdtsxI
qmg8pObu2s7z+BqC0iM5z01deygAfgP4NRzmQqvECiDmjKWxXZlzQNPxnlu3MJZMrfDXTSzD
eIBph1YOIag==";
```

// Private key from appendix B

```
  KeyFactory keyFactory = KeyFactory.getInstance("RSA");
  RSAPrivateKey privateKey = (RSAPrivateKey) keyFactory.generatePrivate(
    new PKCS8EncodedKeySpec(Base64.decodeBase64(privateK)));


  String jwt = JWT.create()
    .withIssuedAt(new Date())
    .withJWTId(UUID.randomUUID().toString())
    .withSubject(subject)
    .withIssuer(issuer)
    .withAudience(audience)
    .withClaim("resource_id", resourceId)
    .withClaim("email", email)
    .withClaim("first_name", firstName)
    .withClaim("middle_name", middleName)
    .withClaim("last_name", lastName)
    .withExpiresAt(new Date(System.currentTimeMillis()+5*60*1000))
    .sign(Algorithm.RSA256(null, privateKey));
```

```
  System.out.println(jwt);
 }
}
```

Example 2: Validate a SNS Launch message

This example is more complicated, mostly because the auth0 JWT library has no helper method for selecting the right algorithm from the JWT header.

```
import com.auth0.jwt.JWT;
import com.auth0.jwt.algorithms.Algorithm;
import com.auth0.jwt.interfaces.DecodedJWT;
import org.apache.commons.codec.binary.Base64;

import java.security.KeyFactory;
import java.security.NoSuchAlgorithmException;
import java.security.interfaces.ECPublicKey;
import java.security.interfaces.RSAPublicKey;
import java.security.spec.InvalidKeySpecException;
import java.security.spec.X509EncodedKeySpec;

public class JwtProviderExample {
 public static void main(String[] args) throws Exception {
  String token = args[0];

  // Get the algorithm name from the JWT.
  String algorithmName = JWT.decode(token).getAlgorithm();
  // Get the issuer name from the JWT.
  String issuer = JWT.decode(token).getIssuer();

  // Lookup the issuer.
  String publicK = getPublicKeyForIssuer(issuer); // Public key from
appendix A

  // Get the algorithm from the public key and algorithm name.
  Algorithm algorithm = getAlgorithm(publicK, algorithmName);

  // Decode and verify the token.
  DecodedJWT jwt = JWT.require(algorithm)
    .withAudience("therapieland.nl") // Make sure to require yourself to
be the audience.
    .build()
    .verify(token);

  // Read the parameters from the jwt token.
  String subject = jwt.getSubject();
  String resourceId = jwt.getClaim("resource_id").asString();
  String email = jwt.getClaim("email").asString();
  String firstName = jwt.getClaim("first_name").asString();
```

```
  String middleName = jwt.getClaim("middle_name").asString();
  String lastName = jwt.getClaim("last_name").asString();

  System.out.println(String.format("The SNS launch recieved the user
with id %s for resource %s, the user email is %s, the user is known as
%s %s %s.",
    subject,
    resourceId,
    email,
    firstName,
    middleName,
    lastName));
 }

 /**
  * This method should lookup the public key configured with the issuer
from the configuration
  * and / or persistent storage.
  *
  * @param issuer the issuer from the JWT token.
  * @return a public key encoded as String
  */
 private static String getPublicKeyForIssuer(String issuer) {
  // Return the test key from Appendix A.
  return "..." ;
 }

 /**
  * Unfortunately, this implementation of JWT has no helper method for
selecting the right
  * algorithm from the header. The public key must match the algorithm
type (RSA or EC), but
  * the size of the hash algorithm can vary.
  *
  * @param publicKey
  * @param algorithmName
  * @return in instance of the {@link Algorithm} class.
  * @throws NoSuchAlgorithmException
  * @throws InvalidKeySpecException
  * @throws IllegalArgumentException if the algorithmName is not one of
RS{256,384,512} or ES{256,384,512}
  */
 private static Algorithm getAlgorithm(String publicKey, String
algorithmName) throws NoSuchAlgorithmException, InvalidKeySpecException,
IllegalArgumentException {
  switch (algorithmName) {
   case "RS256": {
    return Algorithm.RSA256(getRsaPublicKey(publicKey), null);
   }
   case "RS384": {
```

```
    return Algorithm.RSA384(getRsaPublicKey(publicKey), null);
   }
   case "RS512": {
    return Algorithm.RSA512(getRsaPublicKey(publicKey), null);
   }
   case "ES256": {
    return Algorithm.ECDSA256(getEcPublicKey(publicKey), null);
   }
   case "ES384": {
    return Algorithm.ECDSA384(getEcPublicKey(publicKey), null);
   }
   case "ES512": {
    return Algorithm.ECDSA512(getEcPublicKey(publicKey), null);
   }
   default:
    throw new IllegalArgumentException(String.format("Unsupported
algorithm %s", algorithmName));
  }

 }

 /**
  * Parses a public key to an instance of {@link ECPublicKey}.
  *
  * @param publicKey the string representation of the public key.
  * @return an instance of {@link ECPublicKey}.
  * @throws NoSuchAlgorithmException
  * @throws InvalidKeySpecException
  */
 private static ECPublicKey getEcPublicKey(String publicKey) throws
NoSuchAlgorithmException, InvalidKeySpecException {
  KeyFactory keyFactory = KeyFactory.getInstance("EC");
  return (ECPublicKey) keyFactory.generatePublic(
    new X509EncodedKeySpec(Base64.decodeBase64(publicKey)));
 }

 /**
  * Parses a public key to an instance of {@link RSAPublicKey}.
  *
  * @param publicKey the string representation of the public key.
  * @return an instance of {@link RSAPublicKey}.
  * @throws NoSuchAlgorithmException
  * @throws InvalidKeySpecException
  */
 private static RSAPublicKey getRsaPublicKey(String publicKey) throws
NoSuchAlgorithmException, InvalidKeySpecException {
  KeyFactory keyFactory = KeyFactory.getInstance("RSA");
  return (RSAPublicKey) keyFactory.generatePublic(
    new X509EncodedKeySpec(Base64.decodeBase64(publicKey)));
 }
```

```
}
```

Example 3: Generate a RSA key pair

```
import java.security.*;
import static org.apache.commons.codec.binary.Base64.encodeBase64String;

public class RsaKeyPairGenerator {

 public static void main(String[] args) throws Exception {
  new RsaKeyPairGenerator().generate();
 }

 public void generate() throws NoSuchAlgorithmException {
  // Create a new generator
  KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA");
  // Set the key size
  generator.initialize(2024);
  // Generate a pair
  KeyPair keyPair = generator.generateKeyPair();
  // Output the public key as base64
  String publicK = encodeBase64String(keyPair.getPublic().getEncoded());
  // Output the private key as base64
  String privateK =
encodeBase64String(keyPair.getPrivate().getEncoded());

  System.out.println(publicK);
  System.out.println(privateK);
 }
}
```

Example 4: Generate a EC key pair

```
import java.security.*;
import static org.apache.commons.codec.binary.Base64.encodeBase64String;

public class EcKeyPairGenerator {

 public static void main(String[] args) throws Exception {
  new EcKeyPairGenerator().generate();
 }

 public void generate() throws NoSuchAlgorithmException {
  // Create a new generator
  KeyPairGenerator generator = KeyPairGenerator.getInstance("EC");
  SecureRandom random = SecureRandom.getInstance("SHA1PRNG");
  // Set the key size and random
  generator.initialize(256, random);
  // Generate a pair
  KeyPair keyPair = generator.generateKeyPair();
  // Output the public key as base64
  String publicK = encodeBase64String(keyPair.getPublic().getEncoded());
  // Output the private key as base64
  String privateK =
encodeBase64String(keyPair.getPrivate().getEncoded());

  System.out.println(publicK);
  System.out.println(privateK);
 }
}
```

SNS Launch protocol Python examples

Example 1, generate a SNS Launch token with an RSA key

```
import jwt
import time
from uuid import uuid4


def main():
    # The public key as provided by appendix A.
    private_key = '...'
    # Format as PEM key
    public_key_formatted = f'-----BEGIN PRIVATE KEY-----\n' \
        f'{private_key}' \
        f'\n-----END PRIVATE KEY-----'

    # Time function
    payload = {}
    payload['sub'] = 'urn:sns:user:wikiwijk.nl:123456'
    payload['aud'] = 'therapieland.nl'
    payload['iss'] = 'wikiwijk.nl'
    payload['resource_id'] = 'dagstructuur'
    payload['first_name'] = 'Klaas'
    payload['middle_name'] = 'de'
    payload['last_name'] = 'Vries'
    payload['email'] = 'klaas@devries.nl'
    payload['iat'] = time.time()
    payload['exp'] = time.time() + (5 * 60 * 1000)
    payload['jti'] = str(uuid4())

    jwt_encode = jwt.encode(payload, public_key_formatted,
algorithm='RS256').decode('utf8')
    print(jwt_encode)


if __name__ == '__main__':
    main()
```

Example 2: Validate a SNS Launch message

```
import sys
import jwt


def main(jwt_token):
    # The public key as provided by appendix A.
    public_key = '...'
    # Format as PEM key
    public_key_formatted = f'-----BEGIN PUBLIC KEY-----\n' \
        f'{public_key}' \
        f'\n-----END PUBLIC KEY-----'
    # Use the JWT decode, make sure to set the audience
    jwt_decode = jwt.decode(jwt_token, public_key_formatted,
                            audience="therapieland.nl")
    user_id = jwt_decode['sub']
    email = jwt_decode['email']
    first_name = jwt_decode['first_name']
    middle_name = jwt_decode['middle_name']
    last_name = jwt_decode['last_name']
    issuer = jwt_decode['iss']
    unique_message_id = jwt_decode['jti']
    treatment_id = jwt_decode['resource_id']
    print(f'User {first_name} {middle_name} {last_name} with email
{email} '
          f'from {issuer} wants to launch treatment {treatment_id} '
          f'with launch id {unique_message_id}')
```

Example 3: Generate a RSA key pair

```
from cryptography.hazmat.primitives import serialization as
crypto_serialization
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.backends import default_backend as
crypto_default_backend


def main():
    key = rsa.generate_private_key(
        backend=crypto_default_backend(),
        public_exponent=65537,
        key_size=2024
    )
    private_key = key.private_bytes(
        crypto_serialization.Encoding.PEM,
        crypto_serialization.PrivateFormat.PKCS8,
        crypto_serialization.NoEncryption())
    public_key = key.public_key().public_bytes(
        crypto_serialization.Encoding.PEM,
        crypto_serialization.PublicFormat.SubjectPublicKeyInfo
    )
    print('Public key {}'.format(public_key))
    print('Private key {}'.format(private_key))


if __name__ == '__main__':
    main()
```

Example 4: Generate a EC key pair

```
from cryptography.hazmat.backends import default_backend as
crypto_default_backend
from cryptography.hazmat.primitives import serialization as
crypto_serialization
from cryptography.hazmat.primitives.asymmetric import ec


def main():
    key = ec.generate_private_key(
        curve=ec.SECP256K1,
        backend=crypto_default_backend()
    )
    private_key = key.private_bytes(
        crypto_serialization.Encoding.PEM,
        crypto_serialization.PrivateFormat.PKCS8,
        crypto_serialization.NoEncryption())
    public_key = key.public_key().public_bytes(
        crypto_serialization.Encoding.PEM,
        crypto_serialization.PublicFormat.SubjectPublicKeyInfo
    )
    print('Public key {}'.format(public_key))
    print('Private key {}'.format(private_key))


if __name__ == '__main__':
    main()
```

SNS Launch protocol PHP examples

composer.json

```
{
    "name": "othillo/sns-launch-demo-php",
    "type": "project",
    "require": {
        "lcobucci/jwt": "^3.2",
        "broadway/uuid-generator": "^0.4.0",
        "ramsey/uuid": "^3.8"
    },
    "require-dev": {
        "phpunit/phpunit": "^8.0"
    },
    "license": "MIT"
}
```

```php
<?php

require_once __DIR__ . '/../vendor/autoload.php';

use Broadway\UuidGenerator\Rfc4122\Version4Generator;
use Lcobucci\JWT\Builder;
use Lcobucci\JWT\Signer\Rsa\Sha256;

$signer = new Sha256();

$privateKey = 'MIIEpwIBADANBgkqhkiG9w0BAQEFAASCBJEwggSNAgEAAoH+AL7TOqN8jbMq++PBTATgmItjJSprOnFYP6GbIsIReDN9
scha/BEPngzw8moK370irc8Ei+lmgl7NDmvO0RY3o4juqherjKBnhhNcZT4St+ouW/Iy/899u4dTuhFL7hbSKraY/BBBdIowMnPipI5Frvm
nrEwwd3fzEmAdDq0HA8UxvCGkawakIWnFGqUxHvfkPyBFuEVtJ7HEyFvKqlBOujS7rE8WgB5yV2IInwo2Wnb801xGAxkpUhTevCGnnYlycj
3RvLxRmgPQi2VGhd1oGaZA5FZGubZILWtG74oWJ3OPF7ZxGIVVeq2Gp7VO2bb1q1R5V+0mESBvtSnEok8CAwEAAQKB/VO7cg6Mt8y3fsHIb
qfxOV5oScWcOY/Erl8mKJFJgxns/JayvcpqtOpuy6AWV2ixj9y33QC0V15r0fkiTgLWtS5/sykhwFoeMunJ8C7VndfnMbdMA42zWRcfeRTf
4YAoBlALPwePASklzu2ktJotH4MyvNrNpY5/nT+JYIgx/LxhIwk/HxJ6uVYiFpAINfAGfBphcgxzKWnV23WvRYtrIJc/XXLvSxK08tvoZfm
4c4quf1i3LpTc+1mZmT+jefZoXQcWUnEbCk5Q/8gvDigHMbdOlTqT4/iNj/03PmueWsljiyhbXDYOVGJCaGQpeNaFnhilXPrYEBkAvXIOg6
ECfw7l7td0wyPP0vCYFcbQEr3qng9vg2ISVas8gIOU/OeKNSJ9+wbKWcd0DAztxGShuqDZjBXj+RSEL1XrABjDpk9RqpgkBx3NNXEbCBnYg
3+LU8HCtUBWi5amaJi8JH2839cVXjdZbPXBPmp5S93SKjmuoiBas8oKITh0yEwwdb8CfwzPAeg765BhD4AmwSzoQRy6Sfxf6R0Z8Uo9a2mx
BiGSKPvX7zQMG384208FvTlaW3UoOAhSN6HsfBwWT9pzRIaWAkFP8CWxRiRqzg20FYzTweQZOnqje6YRYSocX64l22zhqV3Y3DdqevIiGpx
DFqFM8QXeaAcchCvg6LpTl3ECfwqlC1RynwM1eLhjUhvti5aazjilKrCl/QQOhJx/lXwyaeitLvEZH7C9H+cU8+AbFmfbSJZTfyLDl7bB5B
3NnUTLSyLNizAl8WtRLyaYZsx41m15G1xO+gm3+MA4nbIhg6YAJINTp+CoJFqbNDPX+EeimUCYziErv7TA7GRTs60Cfws28F+KnzzBjtXQm
NCd5eymOwNKYovFXBt5XWOjyE96boHa1ahHdYfVm0c8KipeL7eLaEv42JbgvOXGr1IAHJ6OFxliSUxnQ5e9H/6ljzzHZ3s0j5wzKZ8EloNN
ZoTOxqk1h5oQtveaNl1seMoaf2TpPhq6WXDoidz1Ri9l4zECfmzg4k6Jo2YpZVAm1xQU5SPYDawH4DNlWeTMnqBEwfZap7wu79zJkZYdCae
gzabb/FxFSu0+21djZbq4+PdtsxIqmg8pObu2s7z+BqC0iM5z01deygAfgP4NRzmQqvECiDmjKWxXZlzQNPxnlu3MJZMrfDXTSzDeIBph1Y
OIag==';
$privateKeyPem = "-----BEGIN RSA PRIVATE KEY-----\n" . $privateKey . "\n-----END RSA PRIVATE KEY-----";

$token = (new Builder())
->setIssuer('wikiwijk.nl')
->setAudience('therapieland.nl')
->setId((new Version4Generator())->generate(), true)
->setIssuedAt(time())
->setExpiration(time() + 5*60)

->set('sub', 'urn:sns:user:wikiwijk.nl:123456')
->set('resource_id', 'dagstructuur')
->set('first_name', 'Klaas')
->set('middle_name', 'de')
->set('last_name', 'Vries')
->set('email', 'klaas@devries.nl')

->sign($signer, $privateKeyPem)
->getToken();

echo $token;
```