SWITCHING PROGRAMME

CSS SECURITY AND API SUPPORTING INFORMATION

Version: **1.0**
Status: **Final**
Author: Stuart.Pitcher@smartdcc.co.uk

Product Description History

Version	Date	Author	Summary of Changes
0.1	18 Feb 2022	Stuart Pitcher	First Draft
0.2	01 Mar 2022	Stuart Pitcher	Updated after review
1.0	06 May 2022	Stuart Pitcher	Updated based on REC comments

Product Description Approval – Contract Acceptance

Version	Date	Name	Title / Responsibility

Product Description Approval – Programme Approval

Version	Date	Name	Title / Responsibility

DCC Controlled

Table of contents

1 Introduction 6
1.1 Purpose 6
1.2 Scope 6
1.3 Document Structure 6
1.4 Referenced Documents 7

2 CSS Interface Description 8
2.1 Environment 8
2.2 CSS Interface Security Properties 9
2.3 Communications Channels 9
2.3.1 Internet Access to CSS 10
2.3.2 Private Network Access to CSS 10
2.3.3 Access to CSS using a CSS Interface Provider 11
2.3.4 Multi-Channel Access to CSS 12
2.4 Security Requirements 13
2.4.1 Transport Layer Security 13
2.4.2 JSON Message Signing 13

3 Public Key Infrastructure 14
3.1 Overview 14
3.2 Digital Certificates 14
3.3 What is meant by a Public Key? 14
3.4 Certificate Authority 15
3.4.1 Certificate Policy 15
3.4.2 Certificate Usage 16
3.5 Determining Trust and Validity 16
3.5.1 Chain of Trust 16
3.5.2 Certificate Revocation Lists 17
3.5.2.1 Overview 17
3.5.2.2 Revocation Checking Requirement for Relying Parties 17
3.6 What SWIKI Certificates Do I Need? 18
3.6.1 Direct (Internet) Access to CSS 19
3.6.2 CSS Interface Provider Access to CSS 20
3.7 Key Management 20
3.7.1 Overview 20
3.7.2 CSS Interface Provider 21
3.8 Frequently Asked Questions 22

4 Transport Layer Security 23
4.1 Overview 23
4.2 CSS Communications Security Requirements 23
4.3 TLS Requirements and Configuration 23
4.3.1 TLS Key Generation and Certificate Signing Requests (CSRs) 23
4.3.2 Certificate Issuance 24

5 JSON Message Signing 25
5.1 CSS Message Authentication Requirements 25
5.2 JWS Configuration Requirements 25
5.2.1 JSON Schema Definition 25
5.2.2 JSON Web Signature (JWS) 25

DCC Controlled

5.2.3 Flattened JWS JSON Serialisation Syntax 26
5.2.4 Signing JSON Messages 28
5.2.5 Verifying Signatures 28
5.2.6 Signature Key Generation and Certificate Signing Requests (CSRs) 28
5.2.7 Certificate Issuance 29

6 Other Information 30
6.1 Certificate Signing Requests (CSRs) 30
6.2 Repository Management 30
6.3 Private Keys 31
6.4 Encryption Secrets/Password Guidance 31

7 CSS API Supporting Information 32
7.1 What is REST? 32
7.2 OpenAPI Specification 32
7.3 OpenAPI Initiative 32
7.4 Interface Specification 32
7.4.1 High Level CSS Structure 32
7.5 Webhooks 33
7.5.1 What they are 33
7.5.2 Why do we need them? 33
7.5.3 Subscribing 33
7.5.4 Messages 34
7.5.5 Securing 34
7.5.6 Delivery and Throttling 34
7.5.7 AddressBase® Premium 34
7.5.8 OpenAPI Specification 35
7.5.9 Versioning 35
7.5.10 Generic Asynchronous Message Flow 36
7.6 CSS API specification 36
7.6.1 General Principles 36
7.6.2 POST 37
7.6.3 PATCH 37
7.6.4 PUT 37
7.6.5 CorrelationId and EventId 37
7.6.6 RegistrationId 38
7.6.7 Standard Response Body – for information 39
7.7 Error Handling 39
7.7.1 Synchronous Errors 39
7.7.2 Asynchronous Errors 40
7.7.3 Validation 40
7.7.4 Errors Payload – for illustration 40

DCC Controlled

Table of Figures

Figure 1 - CSS Interface Context 8
Figure 2 - Internet Access to CSS 10
Figure 3 - Private Network Access to CSS 11
Figure 4 - CSS Interface Provider 11
Figure 5 - Digital Certificate 14
Figure 6 - Signing and Verifying Signatures 15
Figure 7 - Certificate Authority 16
Figure 8 - Certificate Chain 17
Figure 9 - CRL 18
Figure 10 - Direct (Internet) Access 19
Figure 11 - CSS Interface Provider 20

Table of Tables

Table 1 - Security Services & Mechanisms 13
Table 2 – Digital Certificate Requirements by Party Type 19
Table 3 - FAQs 22
Table 4 - PoCs .... **Error! Bookmark not defined.**
Table 5 - CSR Guidance 24
Table 6 - RFC 7515 Logical Values 26
Table 7 - JWS Specification 27
Table 8 - Signing a JSON Message 28
Table 9 - Verifying JWS Signatures 28
Table 10 - CSR Guidance 29

DCC Controlled

1 Introduction

1.1 Purpose

This document provides supporting information for the Centralised Switching Service (CSS) Interface, defining the interface usage requirements and responsibilities for Market Participants to securely exchange information.

1.2 Scope

The scope of this document is to define the operational context and constraints in which the CSS Interface operates to provide consistent performance including:

- Interaction with multiple environments;
- Operational and Security responsibilities;
- High level descriptions of operational interfaces.

This document also defines the operational procedures required by CSS Users to securely exchange registration and address information with CSS using Public Key Infrastructure (PKI) Certificates.

1.3 Document Structure

This document is structured as follows;

Section 1 **Introduction**: this section;

Section 2 **CSS Interface Description** provides an overview of the interface;

Section 3 **Public Key Infrastructure** provides an explanation of, and guidance on the use and management of Public Key Certificates and associated keys;

Section 4 **Transport Layer Security (TLS)** details the processes for generation, distribution, use and management of TLS keys and Certificates;

Section 6 **JSON Message Signing** details the processes for generation, distribution, use and management of JSON message signing keys and Certificates;

Section 6 **Other Information** describes the processes and procedures for distributing key cryptographic key material, including CSRs and Certificates;

Section 7 **CSS API Supporting Information** provides explanations and guidance for the content set out in the CSS Developer Portal.

DCC Controlled

1.4 Referenced Documents

Key	Title	Issue
[1]	CSS Developer Portal - https://devportal.centralswitchingservice.co.uk/	4.1
[2]	SWIKI Certificate Policy - https://emar.energycodes.co.uk	v1.0
[RFC7515]	JSON Web Signature (JWS) – https://datatracker.ietf.org/doc/html/rfc7515	
[RFC5280]	X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile - https://datatracker.ietf.org/doc/html/rfc5280	
[RFC4627]	The application/json Media Type for JavaScript Object Notation (JSON) - https://datatracker.ietf.org/doc/html/rfc4627	
	REC Data Specification	
	REC Schedule – Central Switching Service Schedule	
	REC Service Definition – Central Switching Service Certificate Authority Service Definition	
	CSS Certificate Profile Specification (Excel) – https://emar.energycodes.co.uk	

2 CSS Interface Description

2.1 Environment

The Centralised Switching Service (CSS) is a cloud-based service hosted on the Microsoft Azure Platform. All interaction with the CSS is achieved through real-time messaging, based on a pre-defined JavaScript Object Notation (JSON).

Figure 1 below shows the CSS Interface in the context of the components that it directly interfaces with:

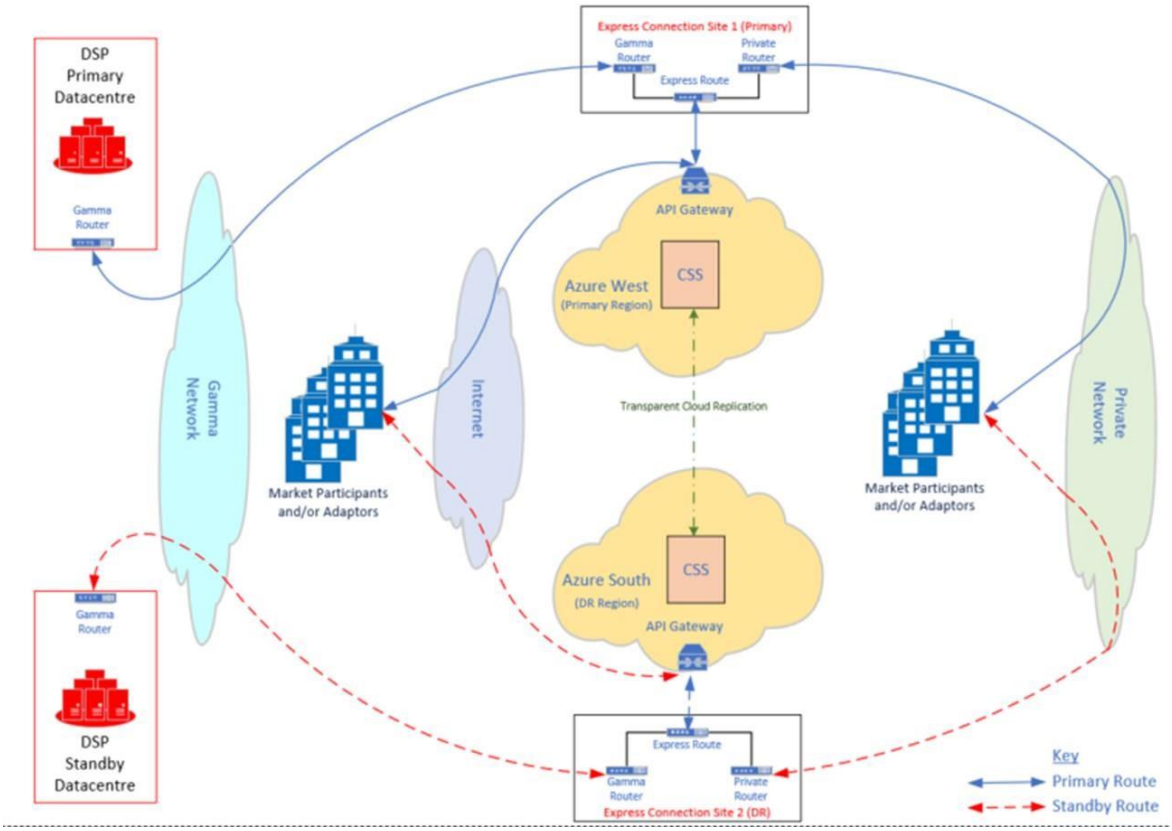


Figure 1 - CSS Interface Context

CSS utilises Microsoft's Azure public cloud offering. To maintain a geographically redundant service, the Azure execution venue takes advantage of Microsoft's UK West and UK South Data centres. This configuration provides the ability to switch between primary and secondary locations with minimum impact and downtime. It also provides the capability to failover from one to the other in the event of a catastrophic failure.

The CSS platform is protected against catastrophic failures pertaining to the hosted components that make up the entirety of the service. This includes both the Addressing and Registration modules. The granular monitoring of each of the applications and underlying infrastructure is paramount to providing a continuous, truly highly available platform.

CSS uses Azure API management to create and maintain interfaces into the CSS systems, using either the Internet or a Private Network via ExpressRoute. Outbound requests i.e., Webhooks will be routed to their respective networks independently of API Management. Azure API management standardises inbound interfaces within the CSS solution.

2.2 CSS Interface Security Properties

There are four types of interactions within the end-to-end architecture: Notifications; Synchronisations[1], Updates and Enquiries. All interactions are synchronous, meaning they follow a protocol whereby an acknowledgement of receipt of a message should be received.

The transactions have specific security properties that must be satisfied. These properties are:

- **Integrity of the message** – This property is to ensure that the message received is as it was when sent and has not been accidentally or maliciously changed in transit. For example, a message asking for multiple switches has the potential to switch large numbers of customers in a single transaction. As above, this could lead to loss of personal data and potential failure of switching requests;
- **Data origin (message) authentication** – Because of the threat of a message or information being sent being impersonated by an unauthorised party, data origin authentication (sometimes called message authentication) is the assurance that a given entity is the original source of the received data. The message is authenticated (signed) using a cryptographic mechanism (in this case a cryptographic key). Where required each message is individually authenticated to allow the recipient to verify the claimed identifier of the message;
- **Mutual Authentication** - Both parties engaged in a communication session authenticate each other such that both are assured of the identity and authenticity of the other;
- **Confidentiality** – Confidentiality is a critical security property because data privacy is a constraint for the REC due to legislation such as the General Data Protection Regulation (GDPR).

All synchronous interactions must satisfy both data origin authentication and data integrity. Consequently, all synchronous interactions should be rejected if:

- They are not from an authentic source; or
- They have been changed in transit; or
- They are not aligned to the JSON schema - defined in [1] CSS Developer Portal

Inbound messages are sent to the CSS by calling pre-determined URLs, whilst the outbound messages from the CSS are delivered to a URL (webhook[2]) of the recipient's choice.

2.3 Communications Channels

The communication channel options available to interact with the CSS are:

- Internet Access;
- Private Network Access (Gamma);
- CSS Interface Provider.

All parties are advised to implement TLS connection pooling both to support their own Non-Functional Requirements (NFRs) and to reduce the centralised and accumulated impact of high potentially high traffic volumes on the CSS system.

[1] Synchronisations require confirmation of processing in addition to acknowledgement of receipt.

[2] Webhooks are URLs which are event-based, meaning that when a specific event occurs in CSS, a message can be delivered to a pre-defined URL.

2.3.1 Internet Access to CSS

The default access for all modern and adaptable applications running in the Microsoft Azure Cloud is via the Internet. As part of the core Azure offering, Microsoft will provide a series of high-performing, scalable, and resilient Internet Gateway (IGW) servers. Behind the Azure IGWs will then be a scalable bank of firewalls providing protection to the CSS application and supporting infrastructure. These firewalls will be configured to manage the TLS connections for all inbound and outbound JSON messages.

At the other end of the internet connection will be the Market Participant application domain where the user application resides. Like the Azure environment, if the Market Participant wishes to use the internet for connecting to the CSS then their environment will need to include both a Policy Enforcement Point (PEP) (i.e., firewall) and one or more Internet Gateways depending on their capacity and resilience requirements.

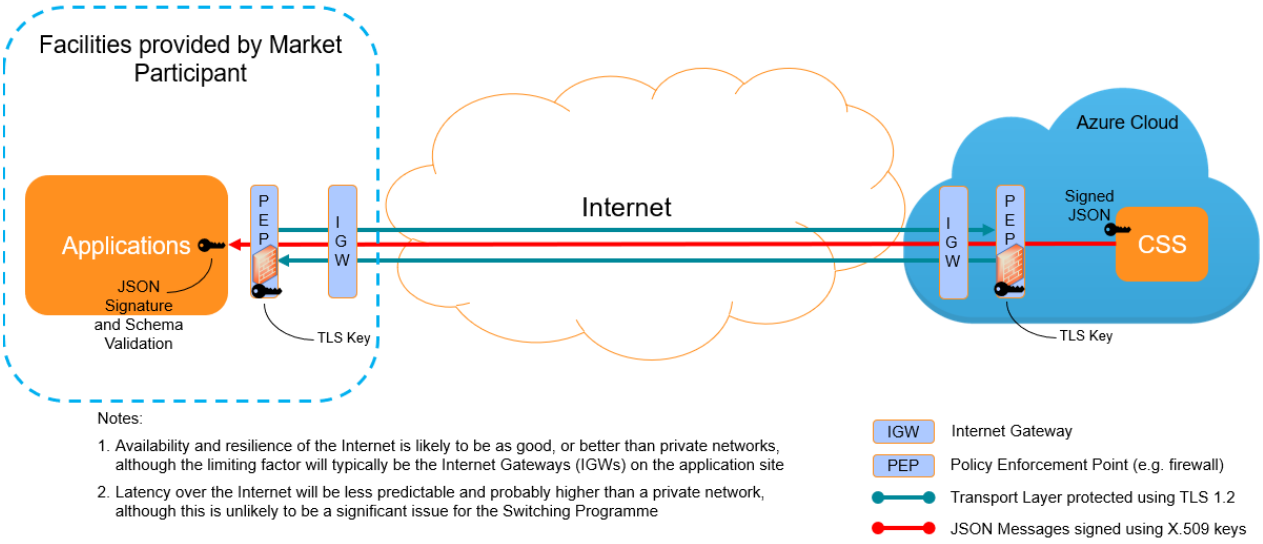


Figure 2 - Internet Access to CSS

2.3.2 Private Network Access to CSS

Microsoft supports access to the Azure cloud via several trusted connection partners who then access the Azure cloud via a Microsoft product called Azure ExpressRoute. ExpressRoute locations are the entry point to Microsoft's network. These locations are where ExpressRoute partners issue cross-connections to Microsoft's network.

Integrating the connectivity layer of Private Networks requires the network provider to physically deploy a boundary gateway (or edge router) into the target datacentre where Private Network cables are terminated. For the Azure environment, these boundary gateways will be deployed in the Azure connection partner's datacentre. The boundary gateways will then be connected to ExpressRoute transit connections from the connection partner to the Azure cloud. Traffic via these ExpressRoutes will then be processed in the Azure cloud in a similar way to traffic over the internet, all of which will be transparent to the CSS application.

In this configuration the encrypted TLS connections will be established between the Policy Enforcement Points (PEP) in the application environment and the PEPs in the Azure cloud, thus resulting in the transport connection being encrypted all the way from the application datacentre, through the private network and Azure ExpressRoute to the Azure cloud.

The approach of using ExpressRoute to connect to the Azure cloud will be standard for all supported Private Networks. However, only the iX network will be available to Market Participants wishing to connect their applications directly to the CSS. This is because iX provides basic IP (layer 3) connectivity and the DTN does not. The DTN while running over an IP network is packaged as a fully managed service which does not support native application access to the underlying IP protocol layer. Therefore, Market Participants wishing to utilise the DTN to interact with the CSS application would need to consider a CSS Interface Provider as described below.

DCC Controlled

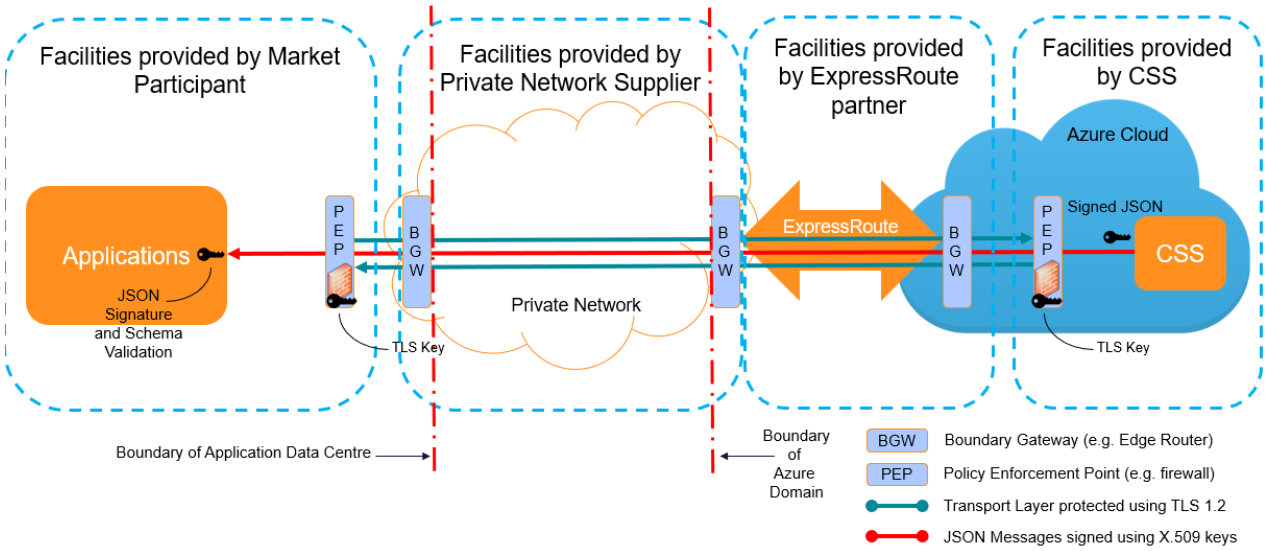


Figure 3 - Private Network Access to CSS

2.3.3 Access to CSS using a CSS Interface Provider

An alternative to the Market Participant applications accessing the CSS directly is the option to route via services provided by a CSS Interface Provider. In this scenario the CSS Interface Provider will offer a connection service over a trusted adaptor interface on its own commercial terms. The exact details of the connection service will be defined by the CSS Interface Provider, but will essentially (as a minimum) need to support the following functionality:

- For messages being sent to the CSS, the CSS Interface Provider will process the inbound application request and then create the required JSON message (including signing the message on behalf of the Market Participant) before sending to the CSS over one of the available access networks. This could either be via the Internet, iX, or DTN. In all cases it is anticipated that the transport mechanism would be largely hidden from the Market Participant behind the managed service.
- Messages originating from the CSS will be received and validated by the CSS Interface Provider before being repackaged and sent to the Market Participant via the custom Switching Adaptor Interface

For the CSS Interface Provider to successfully create CSS JSON messages, if the Market Participant applications have not already done so, the CSS Interface Provider will need to manage the signing keys on behalf of the Market Participant. The process for managing and/or exchanging these keys will need to be agreed with the CSS Interface Provider.

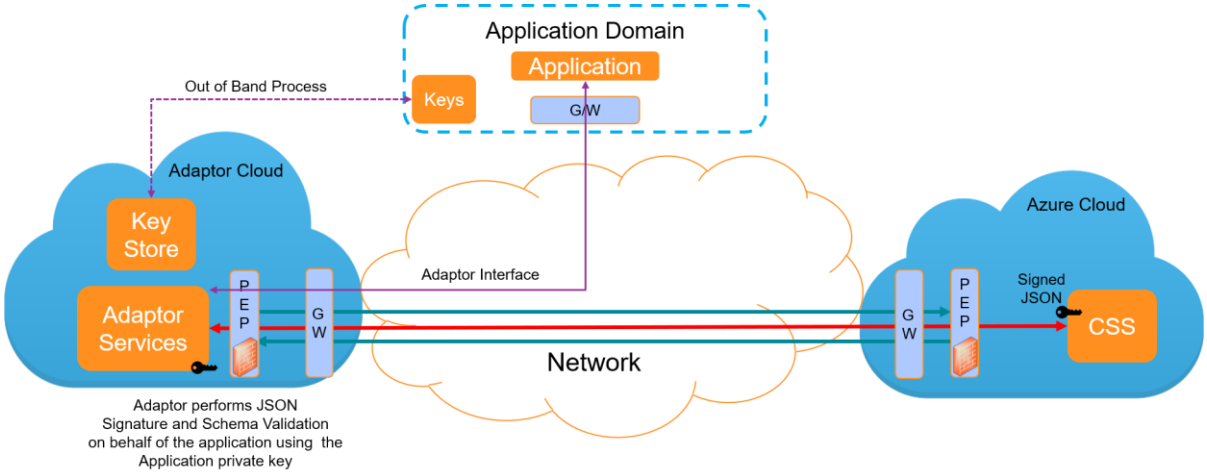


Figure 4 - CSS Interface Provider

2.3.4 Multi-Channel Access to CSS

The CSS has been designed to be agnostic of the underlying access network and will therefore happily interact with any of the three options highlighted above. The only constraint is that once an access route has been selected then all communication between the Market Participant's application and the CSS will be transported over the same access network. However, the CSS solution will also support the evolution of the Market Participant applications allowing them to be modified to use different network options as they develop.

CSS advertises multiple endpoints for receiving messages each of which fulfils a specific business function, for example, a specific endpoint is used to receive switch requests. Multiple external systems with the same business needs will deliver messages to the same endpoint, for example, all switch requests are sent to the same endpoint regardless of which participant instigated the request.

To receive messages from CSS, a recipient registers a target endpoint for each Market Participant role they hold and all outgoing messages applicable to that role are sent to this single endpoint by CSS.

The purpose of the interface is to provide resilient secure wide area network connectivity between the CSS and Market Participant Data Centres.

DCC Controlled

2.4 Security Requirements

The various network access options to CSS are further detailed in the previous section. However, common to all network access options is the requirement for two specific elements of security:

- Transport Layer Security (TLS 1.3 where possible TLS 1.2 as a minimum); and
- JSON Message signing.

2.4.1 Transport Layer Security

Transport Layer Security (TLS) satisfies most of these security objectives. It is set up by configuring Policy Enforcement Points (typically firewalls) at either end of the network connection. This ensures the transfer of the JSON messages across the underlying network is via a secure encrypted channel.

2.4.2 JSON Message Signing

Authentication within the switching network is based on two form factors:

- the first is covered by the establishment of a secure communications channel provided by TLS, as described above;
- the second is the authentication of individual JSON messages which is achieved through the application of a digital signature.

Applying a digital signature to a JSON message also adds a layer of data integrity assurance. Only messages carrying a payload require a digital signature. HTTP responses with no message payload do not require to be digitally signed.

The security requirements and how and where they are satisfied can be seen in the table below:

Objective	Mechanism	Provided By
Confidentiality	Encryption	Transport Layer Security
Data Integrity	Message Authentication Code (MAC)	Transport Layer Security
	Hash Function	
	Digital Signature	JSON Message Signing
Data Origin Authentication	Message Authentication Code (MAC)	Transport Layer Security
	Digital Signature	JSON Message Signing
Non-Repudiation[3]	Digital Signature	JSON Message Signing

Table 1 - Security Services & Mechanisms

The implementation of both Transport Layer Security and JSON Message Signing require the use of Public-Key Cryptography which depends on the use of Digital Certificates. This is described in the next section.

[3] Non-Repudiation is not a specific security requirement but can be realised using a digital signature.

DCC Controlled

3 Public Key Infrastructure

This section provides an overview of Public Key Infrastructure (PKI) and details the PKI requirements and solutions as they relate to Switching. It explains the Digital Certificate requirements for Parties and the interaction between the PKI components and processes, enabling Parties to register, request, install and manage the Certificates necessary to secure the Switching ecosystem.

3.1 Overview

Implementing the mechanisms required to secure the flow of CSS messages with TLS and digital signatures requires Digital Certificates. The Digital Certificates are used to secure communications using public key cryptography schemes.

A Public Key Infrastructure (PKI) is a collection of policies, procedures and technology needed to manage Digital Certificates in a public key cryptography scheme.

3.2 Digital Certificates

The objective of a public key cryptography scheme is trust. A Digital Certificate is an electronic signature from a trusted third party that guarantees the validity and authenticity of a public key.

The Digital Certificate binds an entity, being an institution, a person, a computer program, a web address etc., to its public key.

The most common type of Certificate is the one compliant with the X.509 standard, which allows the encoding of a party's identifying details in its structure.

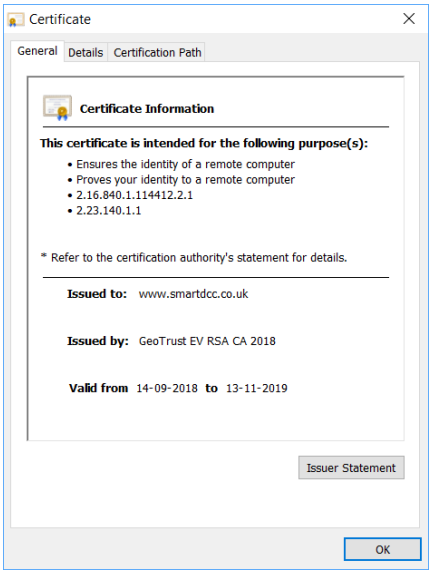


Figure 5 - Digital Certificate

The most common trust model used to validate the validity and authenticity of a public key is a Certificate Authority and this is the trust model used in the Switching System PKI (SWIKI).

3.3 What is meant by a Public Key?

Authentication and message integrity are important concepts in secure communications and are vital security services for Switching.

Authentication requires that entities who exchange messages are assured of the identity that created a specific message. For a message to have "integrity", it should not have been modified during transmission (or at least if it has, you should have a way to find out).

For example, you will want to be sure you are communicating with a real third party (such as CSS) rather than an impersonator. Or if you have received a message from CSS, you will want to be sure that it has not been tampered with by anyone else during transmission.

Traditional authentication mechanisms rely on digital signature schemes that, as the name suggests, allow a party to digitally sign its messages. Digital signatures also provide guarantees as to the integrity of the digitally signed message.

Technically speaking, digital signature schemes require the CSS and each CSS User to hold two cryptographically connected keys: a public key that is made widely available and acts as authentication anchor, and a private key that is used to produce digital signatures on messages. Recipients of digitally signed messages can verify the origin and integrity of a received message by verifying that the attached signature is valid using the public key of the assumed sender.

The unique mathematical relationship between the keys (which is called a trapdoor one-way function) is such that the private key can be used to produce a signature on a message that only the corresponding public key can verify.

It is the public key that is embedded in a Digital Certificate, and the security of public key cryptography relies on ensuring that private keys are kept secure.

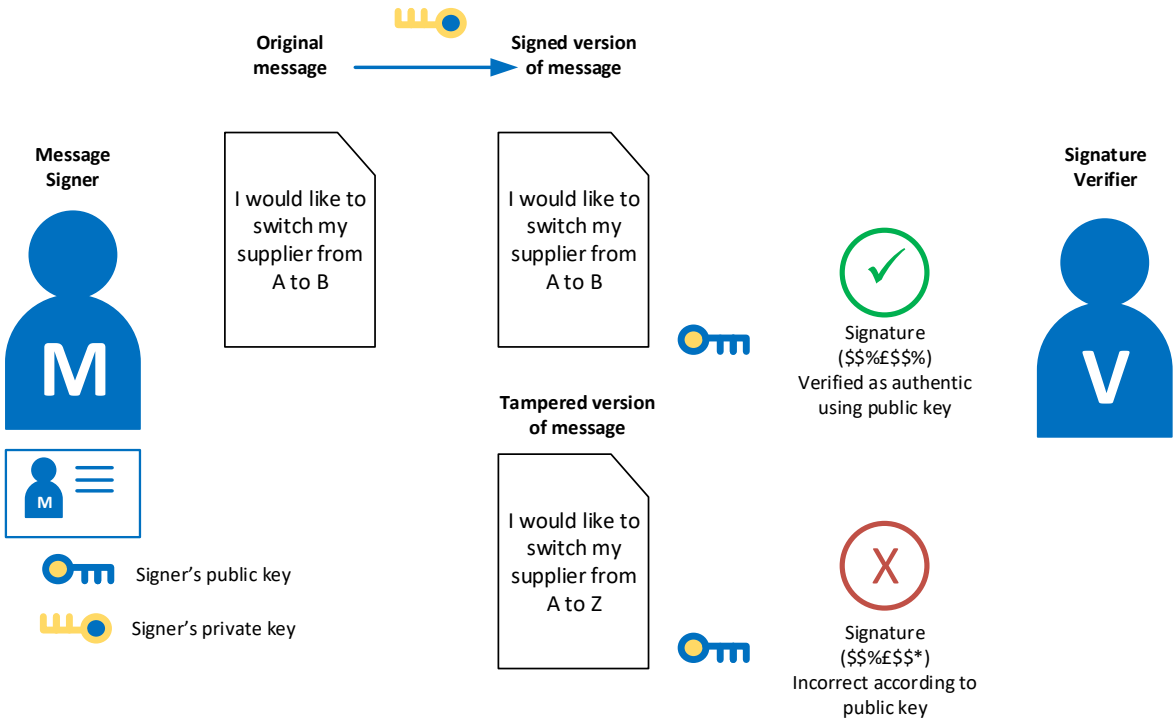


Figure 6 - Signing and Verifying Signatures

3.4 Certificate Authority

3.4.1 Certificate Policy

The Certificate Authority (or CA - synonymous with an Issuing Authority – or IA) is a trusted third party specialised in issuing and managing Digital Certificates. The CSS Certificate Authority (CCA) is managed by DCC. The CCA has published a SWIKI Certificate Policy document which is a named set of rules that indicates the applicability of a Certificate to a particular community and/or class of application with common security requirements, i.e., the Centralised Switching Service. The SWIKI Certificate Policy

DCC Controlled

document is structured in accordance with the guidelines in IETF RFC 3647, with appropriate modifications, deletions, and references to other documentation as appropriate.

For Switching, all Parties shall use Certificates issued by the CCA to secure the interface to CSS for both TLS and digital signatures, except for the Data Service Provider for Smart Metering (DCC SM) and Enduring Change of Supplier Service Provider (ECOS). The interface between CSS, SMDSP and ECOS shall be secured using digital Certificates issued by two existing Certificate Authorities, governed by the Smart Energy Code (SEC):

- To secure TLS connections, certificates must be issued and signed by the DCCKI Certificate Authority;
- Certificates for digital signatures must be issued and signed by the SMKI Certificate Authority.

The CCA will issue digital Certificates to authorised CSS Users (commonly referred to as Certificate Subscribers). These Certificates are digitally signed by the CCA and bind CSS Users with their public keys. As a result, if you trust the CA (and know its public key), you can trust that the specific party's public key included in the Certificate is genuine.

3.4.2 Certificate Usage

Certificates are publicly available as they do not include either the CSS User or the CA's private keys. As such they can be used as anchors of trust for authenticating messages originating from different parties.

CAs also have a Certificate, which they make widely available. This allows the consumers of identities issued by a given CA to verify them by checking that the Certificate could only have been generated by the holder of the corresponding private key (the CA).

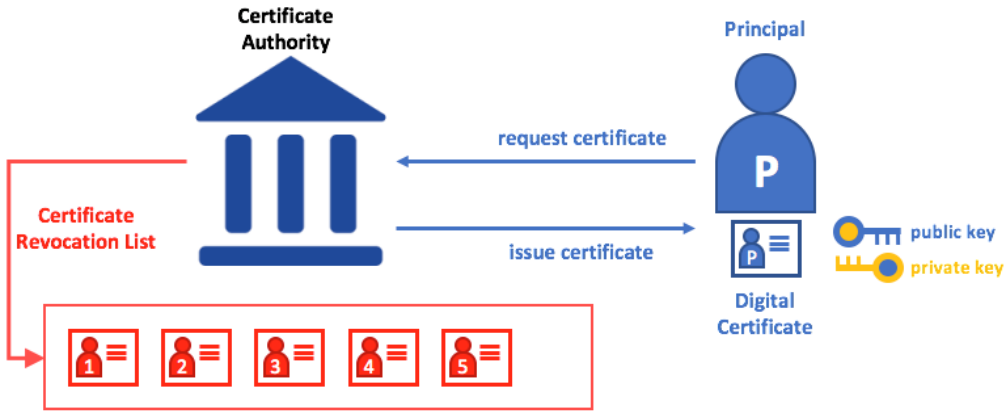


Figure 7 - Certificate Authority

3.5 Determining Trust and Validity

To verify whether a Digital Certificate for an end-entity can be trusted and is valid requires the party doing the validation (or Relying Party) to validate as follows:

3.5.1 Chain of Trust

The standard trust model used in Switching holds that if the Digital Certificate is digitally signed by the CCA, then it can be trusted by the relying party (see section 2.2 of this document). To do this the relying party needs to verify the CCA's Digital Signature and that certain contents of the Digital Certificate are also correct (for example that the Certificate has not expired and that the key usage is for Digital Signatures).

To verify the CCA's Digital Signature, the relying party must have access to the CCA's public key, which itself will be contained in another Certificate signed by a higher-level (Intermediate) or top-level (Root) CA. All Certificates that comprise the 'Certificate

DCC Controlled

Chain' will be required to complete the validation process (also known as 'walking the chain'). All Certificates in the Certificate Chain will be published in a repository accessible by the Switching Community[4].

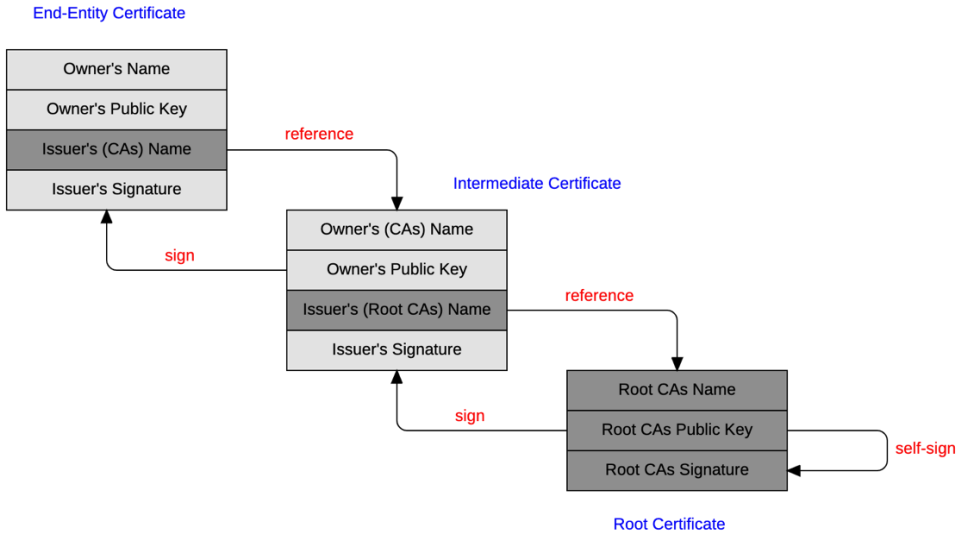


Figure 8 - Certificate Chain

3.5.2 Certificate Revocation Lists

3.5.2.1 Overview

A Certificate Revocation List (CRL) is a list of references to Certificates that a CA knows to be revoked for one reason or another (much like a list of stolen credit cards).

When any party (known as a relying party) wants to verify another party's identity, it first checks the issuing CA's CRL to make sure that the Certificate has not been revoked. A relying party does not have to check the CRL, but if they do not, they run the risk of accepting a compromised identity.

A Subscriber shall ensure that it submits a Certificate Revocation Request in relation to a Certificate in accordance with the REC:

- Immediately upon becoming aware that the Certificate has been compromised, or is suspected of having been compromised; or
- immediately upon ceasing to be an Eligible Subscriber in respect of that Certificate.

3.5.2.2 Revocation Checking Requirement for Relying Parties

Relying Parties are those entities that are using a Certificate to authenticate another Certificate Subscriber named in the Certificate. In the context of CSS, every Certificate Subscriber is also a Relying Party at some point, for example when they are setting up secure channels with TLS or verifying the Digital Signatures on received messages.

All Relying Parties must check the Revocation Status for each Certificate upon which they rely. This is achieved using the Certificate Revocation List that is published at the location defined in the CRL Distribution Point field in every Certificate.

Additional information about CRLs can be found in the [2] **SWIKI Certificate Policy.**

[4] This will be hosted on the Switching Service Management System Interface

DCC Controlled

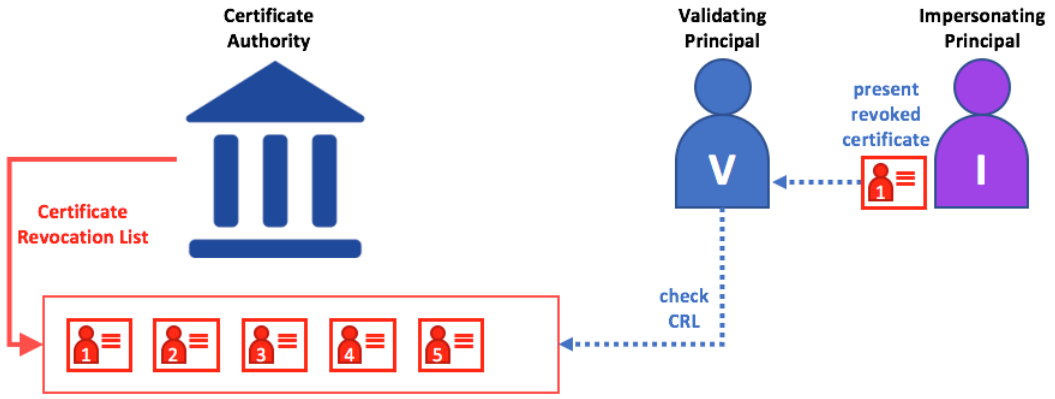


Figure 9 - CRL

3.6 What SWIKI Certificates Do I Need?

The method by which Parties connect to CSS and the resulting architectural pattern introduces some differences in the requirements for certificates and the resulting responsibilities for cryptographic key management. The table below sets out the certificate requirements by Party:

Party	TLS Certificate	Digital Signature Certificate	Notes
CSS	Yes	Yes	The CSS Digital Signature Certificate is made available to the Switching community to allow Parties to verify signed messages from CSS.
CSS Interface Provider[5]	Yes	No – see notes	CSS Interface Provider supporting multiple clients need only a single connection between their service and CSS, secured by their own TLS Certificate. CSS Interface Provider do not require Digital Signature Certificates of their own but must hold a digital signature Certificate on behalf of each client such that they can sign messages on their behalf.
Parties subscribing to a CSS Interface Provider	No	Yes – see notes	Parties subscribing to a CSS Interface Provider do not need TLS Certificates of their own. This is because the connection to CSS is not made directly from their own network but rather from that of the CSS Interface Provider. Though they may proxy much of the responsibility to request and manage Digital Signature Certificates, subscribers retain ownership of, and responsibility for their Digital Signature Certificates.
Parties connecting directly to CSS	Yes	Yes	Parties are wholly responsible for requesting and managing both TLS and Digital Signature Certificates.

[5] CSS Interface Providers typically receive messages from subscribers in one format and transform to JSON format on behalf of the subscriber before forwarding on to CSS.

DCC Controlled

Table 2 – Digital Certificate Requirements by Party Type

At a high level, **TLS Certificates** are required by any Party wishing to connect **directly** to CSS (as well as CSS itself), regardless of whether they are sending messages or receiving messages. This is because the network connection must be secured using **TLS Mutual Authentication** – a protocol where both entities at each end of the connection provide their Certificate for the other entity to authenticate.

Parties subscribing to a CSS Interface Provider do not need TLS Certificates of their own. This is because the connection to CSS is not made directly from their own network but rather from that of the CSS Interface Provider. It is the CSS Interface Provider that must have a TLS Certificate. Further details on how Parties can request a TLS Certificate is provided in **Transport Layer Security**.

Digital Signature Certificates associated with a Party's unique market participant ID (MPID) are required by **every Party** wishing to **send** messages to the CSS as well as the CSS service itself. This includes those Parties opting to use CSS Interface Providers. Further details are provided in **JSON Message Signing.**

Parties connecting directly to CSS (i.e., not via any CSS Interface Provider) require both a TLS Certificate and a Digital Signature Certificate.

Parties **may not** use Certificates for different purposes. For example, a Digital Signature Certificate may not be used for TLS, and vice versa.

Certificates bind to an environment as well as an identify. This means that Parties **may not** re-use Certificates across different environments (SIT, UIT, Pre-Production, Production etc.). For example, Parties cannot use their Production Certificates to secure their UIT environment.

3.6.1 Direct (Internet) Access to CSS

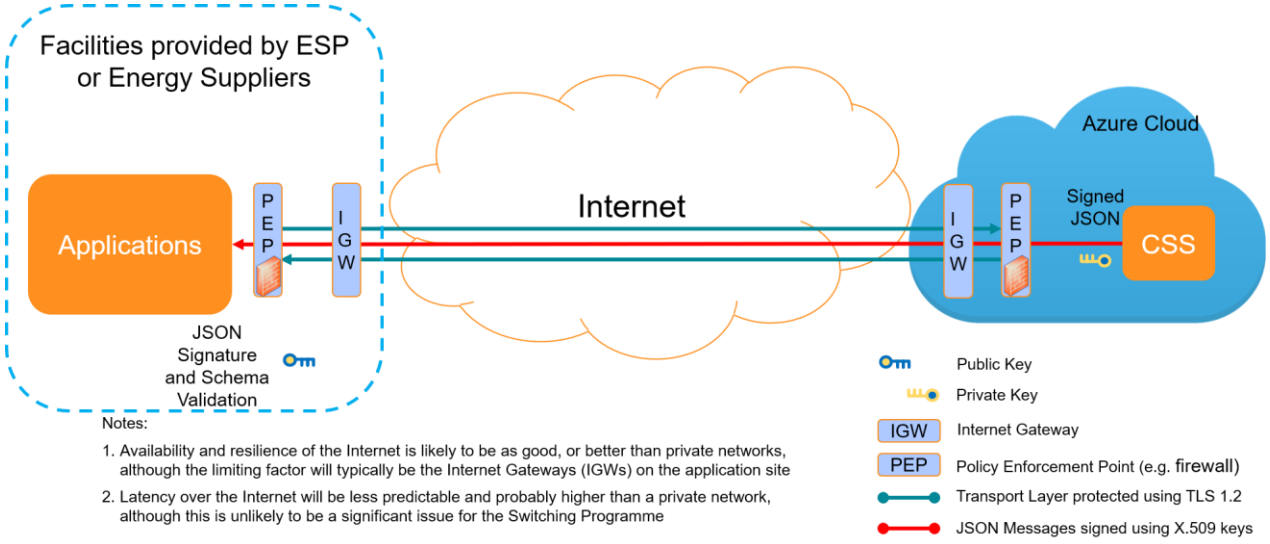


Figure 10 - Direct (Internet) Access

If the CSS User wishes to use the Internet for connecting directly to the CSS, then their environment will need to include both a Policy Enforcement Point (PEP) (i.e., firewall) and one or more Internet Gateways depending on their capacity and resilience requirements.

In **Figure 10 -** above, two Certificates are required:

- A TLS Certificate to secure the channel between their PEP and the Azure cloud;
- A JSON Message Signing Certificate to sign JSON messages that are sent to the CSS.

In this connection scenario, both Certificates must be requested and owned by the CSS User, i.e., the message originator.

DCC Controlled

3.6.2 CSS Interface Provider Access to CSS

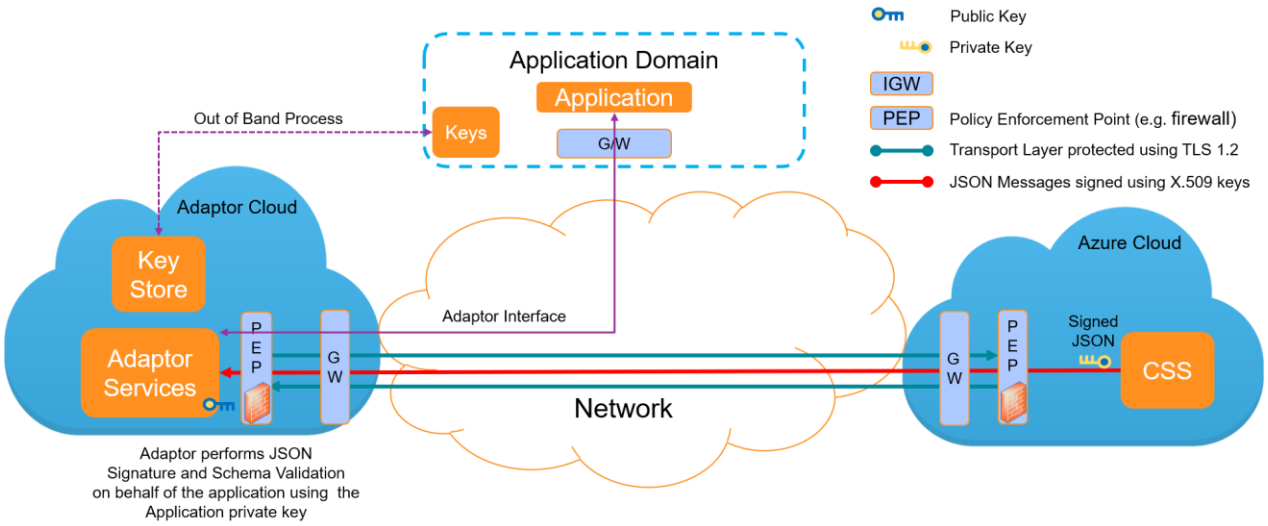


Figure 11 - CSS Interface Provider

An alternative to the Market Participant applications accessing the CSS directly is the option to route via a CSS Interface Provider. In this scenario the CSS Interface Provider will offer a connection service over a trusted adaptor interface on its own commercial terms. The exact details of the connection service will be defined by the CSS Interface Provider, but will essentially (as a minimum) need to support the following functionality:

- For messages being sent to the CSS, the CSS Interface Provider will process the inbound application request and then create the required JSON message (including signing the message on behalf of the Market Participant) before sending to the CSS over one of the available access networks. This could either be via the Internet, iX, or DTN. In all cases it is anticipated that the transport mechanism would be largely hidden from the Market Participant behind the managed service.
- Messages originating from the CSS will be received and validated by the CSS Interface Provider before being repackaged and sent to the Market Participant via the custom Switching Adaptor Interface

In this connection scenario:

- The TLS Certificate must be requested and owned by the CSS Interface Provider;
- The JSON Message Signing Certificate must be owned by the CSS User, i.e., the message originator – not the CSS Interface Provider – but may be requested by the CSS Interface Provider.

3.7 Key Management

3.7.1 Overview

Secure procedures for the handling of cryptographic keys are vital for Cryptosystems. Since Public-Key Cryptography depends on the secrecy of the Private Key, the expectation is that each CSS User should ensure that key material is protected appropriately.

For most organisations, this will involve putting in place a boundary protection device, such as a firewall, that can enforce the controls specified. The device, or Policy Enforcement Point (PEP), must have logical access to a key store for TLS authentication and be able to create or import keys in a secure fashion such as via a Cryptographic Module. What provides the PEP capability, and whether there is a Cryptographic Module in operation and the nature of the Local Public Key Store is at the discretion of the CSS User and subject to their own assessment of risk.

DCC Controlled

Each party shall raise a Major Security Incident as soon as possible using the Switching Portal once they become aware of compromise or suspected compromise of any private key material associated with a Certificate. This may result in the Certificate being revoked.

Where a Certificate is revoked by the CCA, the Certificate will be added to its appropriate X509 Certificate Revocation List, which shall be readily accessible.

Should a Certificate be revoked, the associated public and private keys will no longer be trusted. It is the responsibility of each CSS User to regularly poll the CRL to pick up any Certificate and Certificate Revocation information in a timely fashion.

3.7.2 CSS Interface Provider

Where a CSS Interface Provider is chosen by a Subscriber, key management becomes slightly more complex.

The TLS connection between the CSS Interface Provider and the CSS is secured using the CSS Interface Provider's TLS Certificate. However, JSON messages must be digitally signed using the private key of the message originator, who will not be the CSS Interface Provider. The Digital Signature allows CSS to determine and authenticate the identity of the message originator.

For users of a Service Provider, it is anticipated that the Service Provider will take responsibility for public/private key and Certificate Signing Request (CSR) generation on behalf of the Certificate Subscriber to avoid unnecessary exposure of key material. This does place an extra burden of responsibility upon Service Providers, since they will be responsible for the protection of the private keys of their Service Users, as well as their own. The responsibility for implementing appropriate key management processes – including auditing of these processes – will lie with Service Providers.

DCC Controlled

3.8 Frequently Asked Questions

Question	Response
We already have an SRO/ARO and want to use the same individuals. Can we?	**Yes**. The CSS will acknowledge the existing SROs/AROs, however these will need to be validated by your organisation as indicated above. As the CSS environment is separate from other PKI environments within DCC, this confirmation is required before any Certificates can be issued.
We are a small organisation and do not have a Head of HR. What should we do?	Another officer of the company must validate who you are. The SRO/ARO cannot validate themselves. The Officer can be one that is listed on the Company House Register.
Is the appointment of an ARO mandatory?	**No**. This is an option that is available to the SRO if he/she cannot fulfil the role because of other commitments or other business operational reasons.
Is it mandatory to complete the Technical Contact Environment Authorisation Table?	**Yes**. Forms will be rejected if this table is not complete and correct.
Is the appointment of a Technical Contact mandatory?	It is mandatory only if the SRO is not performing the role. The Technical Contact can be an employee of the Market Participant or a 3rd party of the Organisations' choice, such as a CSS Interface Provider. The Market Participant should ensure the 3rd Party has and maintains appropriate security and controls for the management and security of the MP's Private Key and all supporting Certificate requirements.
We are a 3rd Party that have a group of individuals performing this function for several Market Participants. Can we use a common email to create CSRs and to send/receive Certificates?	**No.** It is mandatory that each person listed on the form must have their own email ID. Any communication without the ID will be rejected and the SRO advised.
How long will it take before we receive confirmation from the SRO process?	If all information in the form is correct, with no omissions and the signed validation letter is received, then you will receive a confirmation from the CSS Certificate Authority immediately thereafter.
Does the SRO need to be aware of all activity surrounding the Certificates?	The SRO, on behalf of the organisation, is responsible for the certs. It is up to the SRO to establish an effective communication process where he/she is kept aware of all activity associated with the certificates, including in the case where there will be a change of Technical Contact.
Who can request Certificates?	Any one of the SRO/ARO/TC. As such it is important to establish good communication such that duplication of cert requests do not occur.

Table 3 - FAQs

4 Transport Layer Security

This section details the processes for generation, distribution, use and storage of TLS keys and Certificates.

4.1 Overview

All communications between the CSS and CSS Users shall be via a secure communications channel. Individual JSON messages will flow over this secure communications channel.

The communications channel will consist of an encrypted and authenticated session between a CSS and CSS User Policy Enforcement Point (PEP). Policy Enforcement Points will typically be boundary firewalls that can negotiate the security parameters required to set-up a secure TLS session.

The TLS session **must** be configured for mutual authentication, such that each entity authenticates the other during the handshake protocol.

4.2 CSS Communications Security Requirements

Connections between the CSS and CSS Users across the switching network are required to be subject to cryptographic protection that ensures the authenticity, integrity, and confidentiality of the connection.

4.3 TLS Requirements and Configuration

Cryptographic protection is applied using TLS, which is a series of protocols designed to make use of TCP to provide a reliable end-to-end secure service.

The secure TLS session will be based on the Transport Layer Security (TLS) v1.2 protocol standard (TLS v1.2 as a minimum) and will make use of authentication using PKCS #3 Ephemeral Diffie Hellman key exchange to generate a shared secret (TLS-RSA) with AES-128-GCM-SHA256 for communications encryption.

If this Authentication step fails, an “HTTP 401 Unauthorized” error will be returned to the CSS User. The error codes are referenced in the Interface Specification [1] CSS Developer Portal.

4.3.1 TLS Key Generation and Certificate Signing Requests (CSRs)

The SWIKI service issues RSA Certificates for TLS – in line NCSC guidance – with the following parameters:

- 2048-bit RSA with SHA256

Public/Private key pairs must be created following successful completion of a CSS User’s registration process. Each CSS User will be responsible for the generation of TLS keys and CSRs used by their message service interface.

Each CSS User will need to issue an associated PKCS #10 Certificate Signing Request’s (CSR) to the CCA to obtain a Public-Key Certificate.

No eligible Subscriber (User) may make a Certificate Signing Request which contains:

- Any information that constitutes a trademark unless it is the holder of the Intellectual Property Rights in relation to that trademark; or
- Any confidential information which would be contained in a Certificate issued in response to that Certificate Signing Request.

DCC Controlled

Where any Certificate Signing Request fails to satisfy the requirements set out in this Section 44, the SWIKI CA:

- Shall reject it and refuse to issue the Certificate which was the subject of the Certificate Signing Request; and
- May give notice to the Party which made the Certificate Signing Request of the reasons for its rejection.

Where any Certificate Signing Request satisfies the requirements set out in this Section 4, the SWIKI CA shall issue the Certificate which was the subject of the Certificate signing Request.

Parties should use the distribution mechanisms for CSRs and Certificates outlined in **Other Information**.

Tools for public/private key pair and CSR generation will depend on the local environment. Representative guidance is available in the following links:

Tool/Platform	Guidance
OpenSSL	https://wiki.openssl.org/index.php/Command_Line_Utilities
Microsoft Azure	https://docs.microsoft.com/en-us/azure/key-vault/certificates/create-certificate-signing-request
AWS	https://docs.aws.amazon.com/cloudhsm/latest/userguide/ssl-offload-windows-create-csr-and-certificate.html

Table 4 - CSR Guidance

4.3.2 Certificate Issuance

On successful verification of a PKCS #10 Certificate request the CA will generate a Public-Key Certificate for the CSS User's Public Key and place that Certificate within a publicly accessible repository.

CSS Users will be able to download required TLS Certificates from the Switching Portal instance.

5 JSON Message Signing

Authentication within the switching network is based on two form factors:

- the first is covered by the establishment of a secure communications channel provided by TLS, as described above;
- the second is the authentication of individual JSON messages which is achieved through the application of a digital signature.

Applying a digital signature to a JSON message also adds a layer of data integrity assurance.

Digital signatures are applied to hashes of JSON messages (sometimes called message digests) and are used to detect unauthorised modifications to data, as well as to authenticate the identity of the signatory. In addition, the recipient of signed data can use the digital signature as evidence in demonstrating to a third party that the signature was, in fact, generated by the claimed signatory.

This section outlines the processes for generation, distribution, use and storage of JSON message signing keys and Certificates.

5.1 CSS Message Authentication Requirements

JSON messages between the CSS and CSS Users across the switching network are subject to cryptographic protection that ensures the authenticity and integrity of each individual message.

This cryptographic protection is applied via the application of a JSON Web Signature (JWS), using a P-256-bit Elliptic Curve Digital Signature Algorithm (ECDSA) key (ref: https://tools.ietf.org/html/rfc7518#section-3.4).

5.2 JWS Configuration Requirements

For most messages sent to or from CSS there is a requirement to apply JWS signatures (the exception being some HTTP responses / acknowledgements which do not need to be signed). Receiving applications must verify the JWS signatures, effectively performing authentication and authorisation of the message sender and confirming that the message has not been tampered with in transit. Verification of each message should be performed using the digital signature Certificate relevant to the environment being used (e.g., SIT, UIT, Pre-Production, Prod etc.).

Even CSS Users that only ever receive notifications from CSS will need to acquire a signing key and associated Certificate. This is because it is a requirement to sign the message which is sent to CSS to register the Market Participant's Webhook.

On receipt of a JSON message and where JSON schema validation is successful, the recipient will authenticate the JSON message by verifying the JWS Digital Signature applied to all JSON messages. This Digital Signature uses a P-256-bit Elliptic Curve Digital Signature Algorithm (ECDSA) key.

JSON messages that fail signature verification will not be processed by the CSS. In this instance, the CSS User will be advised of authentication failure as part of the HTTP Response or Acknowledgment generated and sent to the requesting CSS User, along with an appropriate Response Code.

5.2.1 JSON Schema Definition

JSON message structures are specified through the definition of a JSON schema. The definition of the schema to be used for CSS can be found in the latest REC Data Specification.

5.2.2 JSON Web Signature (JWS)

In the context of CSS, a JSON Web Signature (JWS) is a data structure representing a digitally signed message. The reader is referred to RFC7515, the proposed standard provided by the Internet Engineering Taskforce (IETF) for additional details of the JWS format.

DCC Controlled

A CSS JWS represents digitally signed content using JSON data structures and base64url encoding. A JWS contains the following logical values (each of which is defined in 5.2.3):

Logical Value	Details
JOSE Header	A union of: • JWS Protected Header: Information about the JWS message required to successfully process the signature – cryptographically protected from tampering by the provided JWS signature. • JWS Unprotected Header: Information about the JWS message that has no integrity protection.
JWS Payload	The JSON structured message being secured
JWS Signature	The digital signature over the JWS Protected Header and JWS Payload

Table 5 - RFC 7515 Logical Values

5.2.3 Flattened JWS JSON Serialisation Syntax

Once signed, there are two defined message serialisation formats. Proof of Concept work for CSS has favoured a JSON serialisation referred to as the **Flattened JWS JSON Serialisation Syntax**. This supports a single signature as well as communication of the Certificate and keys used for signing.

There are several fields that may be included in the JWS message within the bounds of the JWS specification. The below specifies which fields are expected to be provided for CSS. The choices have been made to support two main requirements:

- The need to be able to share the details of the Certificate and key used to sign the message; and
- To utilise fields where possible to enable future extension of the defined JWS format.

The flattened JWS JSON Serialisation syntax is based upon the general syntax but flattens it, optimising it for a single digital signature. In summary, the syntax of a JWS using the flattened JWS JSON Serialisation is as follows:

```
{
 "payload":"<payload contents>",
 "protected":"<integrity-protected header contents>",
 "header":<non-integrity-protected header contents>,
 "signature":"<signature contents>"
}
```


DCC Controlled

The JWS specification for CSS is as follows:

Logical Value	Fields	Description/Example Value
Payload	Base64 URL encoded JSON payload using UTF-8 Character encoding	
Protected	The JWS protected header. This comprises several fields, and all fields are integrity protected by the provided signature. As per RFC7515 this is to be encoded as BASE64URL (UTF8(JWS Protected Header)). ***alg****:* the algorithm being used to sign. We must sign with ECDSA using P-256 and SHA-256. "**cty" (Content Type) Header Parameter**: the media type of the secured content (the payload). Allows for inclusion of future enhancements. Currently always "jose+json". ***typ***: the media type of the complete JWS. Used to disambiguate among different types of objects that might be present. Currently always "jose+json".	**"alg":"ES256".** **"cty":"jose+json"** **"typ":"jose+json"** *Example unencoded protected header:* **{"alg":"ES256","cty":"jose+json","typ":"jose+json"}**
Header	The JWS unprotected header. Provides information to indicate which key was used to secure the JWS. It is necessary for the recipient of a JWS to be able to determine the key that was employed for the digital signature. The "**kid**" (key ID) Header Parameter is a hint indicating which key was used to secure the JWS. Contains the **issuer** and **serial number** for the associated Certificate required to identify the public key needed to verify the provided signature. These fields are held within a JSON formatted document.	***iss***: The distinguished name identifying the entity that has signed and issued the Certificate. Must be provided. ***ser***: The unique serial number identifying the Certificate for a given Certificate Authority. Must be provided. [RFC5280] states this is a non-negative integer of length up to 20 octets. Serial Numbers ***must*** be stored and processed as non-negative integers and not, for example, in hex format. *Example base64url encoded unprotected header:* **"{ "iss": "issuer-string", "ser": "1234" }"** Note that quotes must be delimited, since this document is provided as part of the free form alphanumeric field "kid"
Signature	The signature over the base64url encoded payload and JWS protected header – as per RFC7515.	

Table 6 - JWS Specification

5.2.4 Signing JSON Messages

To create a JWS, perform the following steps:

Step	Action
1	Create the content to be used as the JWS Payload.
2	Base64url encode the bytes of the JWS Payload. This encoding becomes the Encoded JWS Payload.
3	Create a JWS Header containing the desired set of header parameters. Note that white space is explicitly allowed in the representation and no canonicalization is performed before encoding.
4	Base64url encode the bytes of the UTF-8 representation of the JWS Header to create the Encoded JWS Header.
5	Compute the JWS Signature in the manner defined for the algorithm being used. The JWS Signing Input is always the concatenation of the Encoded JWS Header, a period ('.') character, and the Encoded JWS Payload. The ***alg*** header parameter MUST be present in the JSON Header, with the algorithm value accurately representing the algorithm used to construct the JWS Signature.
6	Base64url encode the representation of the JWS Signature to create the Encoded JWS Signature.

Table 7 - Signing a JSON Message

5.2.5 Verifying Signatures

When validating a JWS, the following steps MUST be performed. If any of the listed steps fails, then the signed content MUST be rejected:

Step	Action
1	The Encoded JWS Header MUST be successfully base64url decoded following the restriction given in this specification that no padding characters have been used.
2	The JWS Header MUST be completely valid JSON syntax conforming to RFC 4627 [RFC4627].
3	The JWS Header MUST be validated to only include parameters and values whose syntax and semantics are both understood and supported.
4	The Encoded JWS Payload MUST be successfully base64url decoded following the restriction given in this specification that no padding characters have been used.
5	The Encoded JWS Signature MUST be successfully base64url decoded following the restriction given in this specification that no padding characters have been used.
6	The JWS Signature MUST be successfully validated against the JWS Header and JWS Payload in the manner defined for the algorithm being used, which MUST be accurately represented by the value of the ***alg*** header parameter, which MUST be present.

Table 8 - Verifying JWS Signatures

5.2.6 Signature Key Generation and Certificate Signing Requests (CSRs)

The SWIKI service issues Certificates for Digital Signatures – in line NCSC guidance – with the following parameters:

- ECDSA-256 with SHA256 on the P-256 curve

DCC Controlled

Public/Private key pairs must be created following successful completion of a CSS User's registration process. Each CSS User will be responsible for the generation of TLS keys and CSRs used by their message service interface.

Each CSS User will need to issue an associated PKCS #10 Certificate Signing Request's (CSR) to the CCA to obtain a Public-Key Certificate.

Parties should use the distribution mechanisms for CSRs and Certificates outlined in **Other Information.**

Tools for public/private key pair and CSR generation will depend on the local environment. Representative guidance is available in the following links:

Tool/Platform	Guidance
OpenSSL	https://wiki.openssl.org/index.php/Command_Line_Utilities
Microsoft Azure	https://docs.microsoft.com/en-us/azure/key-vault/certificates/create-certificate-signing-request
AWS	https://docs.aws.amazon.com/cloudhsm/latest/userguide/ssl-offload-windows-create-csr-and-certificate.html

Table 9 - CSR Guidance

5.2.7 Certificate Issuance

On successful verification of a PKCS #10 Certificate request the CA will generate a Public-Key Certificate for the CSS User's Public Key and place that Certificate within a publicly accessible repository.

CSS Users will be able to download required CSS Digital Signature Certificates from the Switching ServiceNow instance.

CSS will periodically poll a pre-determined Landmark repository for the latest Digital Signature Certificates for each Party.

A Certificate which has been issued by the SWIKI CA shall be treated as valid for its stated purpose until such time as it is revoked.

The SWIKI CA shall maintain a record of all Certificates which have been issued by it and accepted by a Subscriber (user).

6 Other Information

6.1 Certificate Signing Requests (CSRs)

Though the CSR contains only the public portion of the key pair, a Certificate Authority, by signing a CSR and thus issuing a Certificate, is stating that the contents of the CSR are true. Therefore, it is important that the integrity of the CSR is protected during transmission.

Once a CSR for either a TLS or Digital Signature Certificate has been generated, it will need to be attached to the Request Certificate Service Request in the Switching Service Management Service. The information contained will be validated against the specification as shown in the Certificate Profile

6.2 Repository Management

A Repository is a Participant organisation that holds data in support of PKI operations. This includes policy and related documentation, Certificates and Certificate Status information.

The Repository provides a community-wide accessible mechanism by which primarily Subscribers and Relying Parties can obtain and validate information on Certificates Issued by the CCA/Issuing Authority.

The Issuing Authority is responsible for providing the SWIKI Repository in accordance with the REC.

The SWIKI CA and the SWIKI Certificate Manufacturer each have their own repositories.

The SWIKI Repository associated with the SWIKI Operator Root CA Certificates (SWIKI Certificate Manufacturer) stores the following information:

- All copies of issued SWIKI Operator Root CA Certificates;
- Certificate status and validity meta-data for each SWIKI CA Certificates.

The SWIKI Repository associated with End Entity Certificates shall store the following information:

- All copies of SWIKI End Entity Certificates issued by the SWIKI Operator CA;
- Certificate status and validity meta-data for each SWIKI End Entity Certificate issued;
- The latest version of the SWIKI CRL (Certificate Revocation List);
- All copies of issued SWIKI CA Certificates.

All SWIKI Repositories are subject to access controls using usernames and passwords. Only authorized SWIKI Systems Personnel have access to SWIKI Repositories.

CSRs must be uploaded to and retrieved from the Party's **CSR** folder

Certificates must be uploaded to and retrieved from the Party's **Certificates** folder.

The **CSS Digital Signature Certificate** will be accessible in the Repository by all Parties.

It is the responsibility of each Party to remove the Certificate and/or CSR once signing is complete. DCC will not actively manage these folders, therefore Parties must take due care to ensure their Certificates and/or CSRs do not remain in the Repository longer than they would wish.

DCC Controlled

6.3 Private Keys

For TLS key-pairs, the size of Issuing Authority and any supporting CA-Keys shall be not less than 4096-bit modulus for RSA.

For TLS key-pairs, the size of Subscribers' Private Keys shall be not less than 2048-bit modulus for RSA.

Subscribers (users) are responsible for the Back-Up of their own keys.

Key backups shall at a minimum be protected to the standards commensurate with that stipulated for the primary version of the key.

Subscribers (Subjects) who are natural persons must be authenticated to their cryptographic module before the activation of the Private Key. This authentication may be in the form of a PIN, pass-phrase password, or other activation data. When deactivated, Private Keys must not be exposed in plaintext form.

Unless unavoidable, private keys should never be transmitted. That said, there is a requirement for CSS Interface Providers to hold the private keys of subscribed Market Participants.

Where private keys must be moved, the advice is to export and transport private keys using a PKCS #12 file. A PKCS #12 file - also known as PFX - is a single, password protected Certificate archive that contains the entire certificate chain plus the matching private key.

Window servers have a utility through the MMC that allows the export of an installed TLS server Certificate along with its corresponding private key to a PFX file.

For Linux-based servers you can use OpenSSL to manage Certificates and keys including creating various file bundles including PKCS #12 archives.

For other types of devices, the instructions will vary depending on the type of device you are using. Please consult your documentation.

The PKCS #12 file encryption key should never be used for another transfer, even for a different private key.

6.4 Encryption Secrets/Password Guidance

When creating encryption secrets or passwords to protect key material, Parties are referred to NCSCs guidance on password strategies, and specifically the use of password managers and random words and phrases:

https://www.ncsc.gov.uk/blog-post/three-random-words-or-thinkrandom-0

DCC Controlled

7 CSS API Supporting Information

7.1 What is REST?

- Representational state transfer.
- "*Representational State Transfer (REST) is a software architectural style that defines a set of constraints to be used for creating Web services. Web services that conform to the REST architectural style, termed RESTful Web services (RWS), provide interoperability between computer systems on the Internet*."
- Source: https://en.wikipedia.org/wiki/Representational_state_transfer

7.2 OpenAPI Specification

- V2 was known as Swagger, V3 is now Open API Specification.
  "*The OpenAPI Specification, originally known as the Swagger Specification, is a specification for machine-readable interface files for describing, producing, consuming, and visualizing RESTful web services. Originally part of the Swagger framework, it became a separate project in 2016, overseen by the OpenAPI Initiative, an open source collaborative project of the Linux Foundation. Swagger and some other tools can generate code, documentation and test cases given an interface file.*"
- Source: https://en.wikipedia.org/wiki/OpenAPI_Specification

7.3 OpenAPI Initiative

- "*The OpenAPI Initiative (OAI) was created by a consortium of forward-looking industry experts who recognize the immense value of standardizing on how REST APIs are described. As an open governance structure under the Linux Foundation, the OAI is focused on creating, evolving and promoting a vendor neutral description format.*
  APIs form the connecting glue between modern applications. Nearly every application uses APIs to connect with corporate data sources, third party data services or other applications. Creating an open description format for API services that is vendor neutral, portable and open is critical to accelerating the vision of a truly connected world."
- Source: https://www.openapis.org/about

7.4 Interface Specification

This section describes the interface to the CSS at a high level and how messages are to be composed. The structure of the messages and any specific data validation to be performed is contained in the REC Data Specification and all versions and revisions of the API's are documented in the CSS Developer Portal as OpenAPI Specification Version 3.

https://devportal.centralswitchingservice.co.uk/

7.4.1 High Level CSS Structure

The CSS is a cloud-based service hosted on the Microsoft Azure Platform. All interaction with the CSS will be real-time messaging, based on a predefined JavaScript Object Notation (JSON[1]) format.

Inbound messages will be sent by calling pre-determined URLs whilst the outbound messages are delivered to URLs of the recipient's choice. These services are explained in technical detail in the CSS Developer Portal.

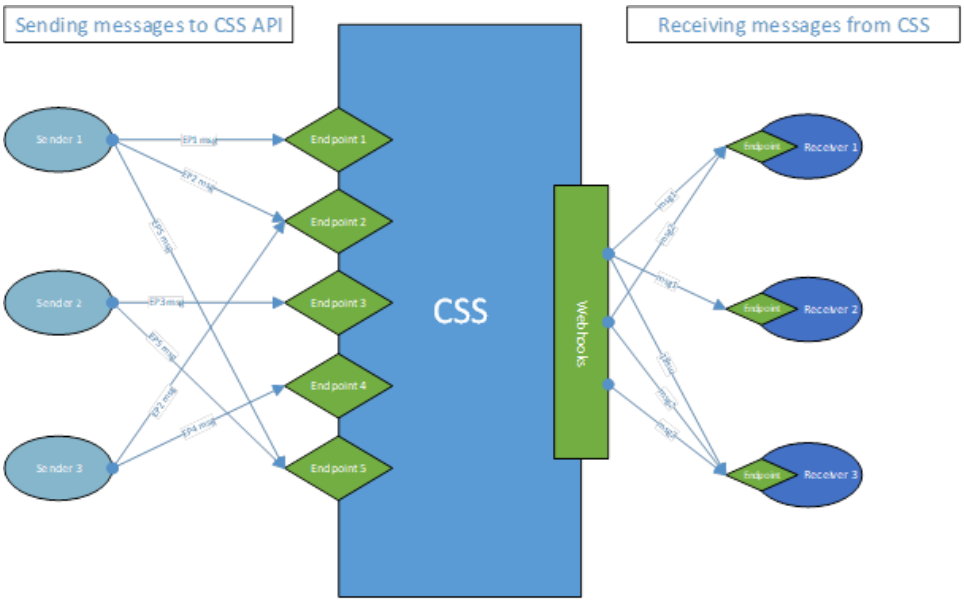
The diagram below depicts the interaction with the CSS from the perspective of sending messages into CSS and receiving messages from CSS.

CSS advertises multiple endpoints for receiving messages each of which fulfils a specific business function, for example, a specific endpoint is used to receive switch requests. Multiple external systems with the same business needs will deliver messages to the

DCC Controlled

same endpoint, for example, all switch requests are sent to the same endpoint regardless of which participant instigated the request.

To receive messages from CSS, a recipient registers target endpoints for each market participant role they hold and all outgoing messages applicable to that role are sent to these endpoints by CSS.



[1] Pronounced Jason, as in "Jason and the Argonauts"

7.5 Webhooks

7.5.1 What they are

As more and more of what we do with connected apps can be described by events, Webhooks are becoming very popular as a mechanism to react to these events in a resource light decoupled way.

A Webhook (also called a web callback or HTTP push API) is a way for an application or system to provide other applications or systems with real-time information. It delivers data to other applications as it happens, meaning you get data immediately, as opposed to typical APIs where you would need to poll for data very frequently (hammer polling) in order to get it real-time. This makes Webhooks much more efficient for both providers and consumers.

Market Participants will need to create an application or system capable of receiving and responding to HTTP requests. CSS will make a HTTP request to your application or system, these will be a POST requests to your Webhook URLs with a JSON body, it is then up to you to process the request and its content.

7.5.2 Why do we need them?

The "messaging" functionality within the CSS is based on **real-time events**, that is, events get triggered by some action within CSS, and these events are then **pushed** out to **subscribers** of those events.

Webhooks give a way for the CSS to push messages out to the various Market Participants who are subscribed to those events in a reactive real-time manner, which makes the system highly performant and scalable.

7.5.3 Subscribing

There will be a way for a Market Participant to request a CSS registration. They will need to provide at least their Market Participant Identifier, their role and company name. Their request will be validated, making sure that they are eligible to be given access to the system. Then a subscription key will be issued to the Market Participant. The key will also be stored within CSS.

DCC Controlled

Once the Market Participant has the subscription key and their Webhooks available, they then need to post to the CSS API as detailed in section 7.6.

Please refer to the CSS Developer Portal for details on the webhook API.

7.5.4 Messages

Once subscribed to the CSS, the market participant will only receive the messages they have subscribed to. They will not receive every possible message sent by CSS. The messages sent out per context type are defined in the CSS Developer Portal.

7.5.5 Securing

Each Market Participant implementing Webhooks has a need to ensure that requests it receives are from trusted parties i.e. the CSS.

When registering, Market Participants can optionally send an x-api-key http header (normally a UUID generated by themselves), this x-api-key http header will be sent back by CSS on all requests to the Market Participants registered Webhook as the same x-api-key http header. This key can be updated by the Market Participant at any time, by re-registering the Webhook with a new x-api-key.

The x-api-key is optional, it is an additional means for Market Participants to ensure that messages came from CSS and for them to be in control of the keys and when they need to rotate them, it can be useful in early development to connect to an API before network security is in place. Best practice would be to have two keys (Primary and Secondary) you would check both against the x-api-key sent in the header. So, let's say the Primary key is registered against your webhook, when you need to rotate the Primary key, register the Secondary key against your webhook, then change the Primary key to a new value, rinse and repeat when rotating the Secondary key.

7.5.6 Delivery and Throttling

CSS is hosted on Azure as an Integration Platform As A Service (IPAAS) in a serverless environment, which allows, in theory, infinite scale out where resources "spin" up and down according to demand.

Event delivery will be via Azure Event Grid. If Event Grid cannot confirm that an event has been received successfully by the subscriber's Webhook, it tries to redeliver the event for a certain amount of time (24 hours by default) using a back off/retry mechanism. After that time is breached, Event Grid will move that event to a "Dead Letter" queue, which will be processed independently by CSS depending on the failure. For more information, see Event Grid message delivery and retry.

To ensure that the CSS does not flood the individual Market Participants with a high volume of events in the cases where the Market Participant Webhook may have been offline (planned maintenance, in a faulted state etc). Market Participants can return a 429 or 503 status code with a **Retry-After** header, CSS will queue event delivery until that point.

The **Retry-After** response HTTP header indicates how long the user agent should wait before making a follow-up request. The two main use cases supported are:

- When sent with a 503 (Service Unavailable) response, this indicates how long the service is expected to be unavailable.
- When sent with a 429 (Too Many Requests) response, this indicates how long to wait before making a new request.

Examples

Retry-After: 2019-08-24T22:57:49.000 (date)

Retry-After: 120 (seconds)

7.5.7 AddressBase® Premium

The Registration and Addressing elements of the CSS will make use of Ordnance Survey AddressBase® Premium (ABP) data (https://www.ordnancesurvey.co.uk/business-and-government/products/addressbase-premium.html).

DCC Controlled

The data passed across the interfaces may only be used for the permitted purpose under the licence with Ordnance Survey. The permitted purpose as defined in the agreement is for the "purposes of enabling switching including (without limitation) for the purposes of design, development, testing, integration and live operational use."

The field mappings contained within this document are taken from v2.5 of the ABP technical specification, which can be found here:

https://www.ordnancesurvey.co.uk/documents/product-support/tech-spec/addressbase-premium-technical-specification.pdf

7.5.8 OpenAPI Specification

Access to the underlying CSS services will be controlled via REST API's fronted by Azure API Management.

The CSS will be using OpenAPI specification version 3.

https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.0.md.

7.5.9 Versioning

New functionality may be required, or bugs may be found. These changes will be submitted to the Retail Energy Code Company (RECCo) and may result in changes being made to the CSS. In order to rollout those changes, it may be necessary to issue a new version of the API.

New versions can be as a result of changes to the API itself, or more likely to underlying business logic directly referred to by the API.

New versions will only be issued if the result of a change is deemed to be a breaking change, non-breaking changes will be issued as revisions to the existing version.

Breaking changes are defined as those caused by:

- Addition and removal of mission critical properties to the schema model, for example, new properties critical to the business process to ensure data integrity.
- Removal of API endpoints (still being consumed).
- Changing schema model properties (optional, max length, datatype etc) for existing properties.
- New domain values. For example, adding a new Status type. In general, this would be deemed a breaking change as consumers will not be expecting this.

Other changes would be defined as non-breaking.

A Market Participant can opt into new version changes when the Market Participant is ready. When every Market Participant is consuming the new version, the old version will be deprecated. There will be a time constraint on this, and each Market Participant will be notified when this happens.

Versioning of the API's will be handled on the API route, i.e.-<CSS URL>/< CSS API Route>/< CSS API Version>/< CSS Operation Route>. Please see below for examples: -

https://centralswitchingservice.com/dccsm/v1.0/rmps/electricity *https://centralswitchingservice.com/dccsm/v1.0/rmps/gas*
https://centralswitchingservice.com/supplier/v1.0/registrations
https://centralswitchingservice.com/supplier/v1.0/registrations/switch
https://centralswitchingservice.com/supplier/v1.0/errors

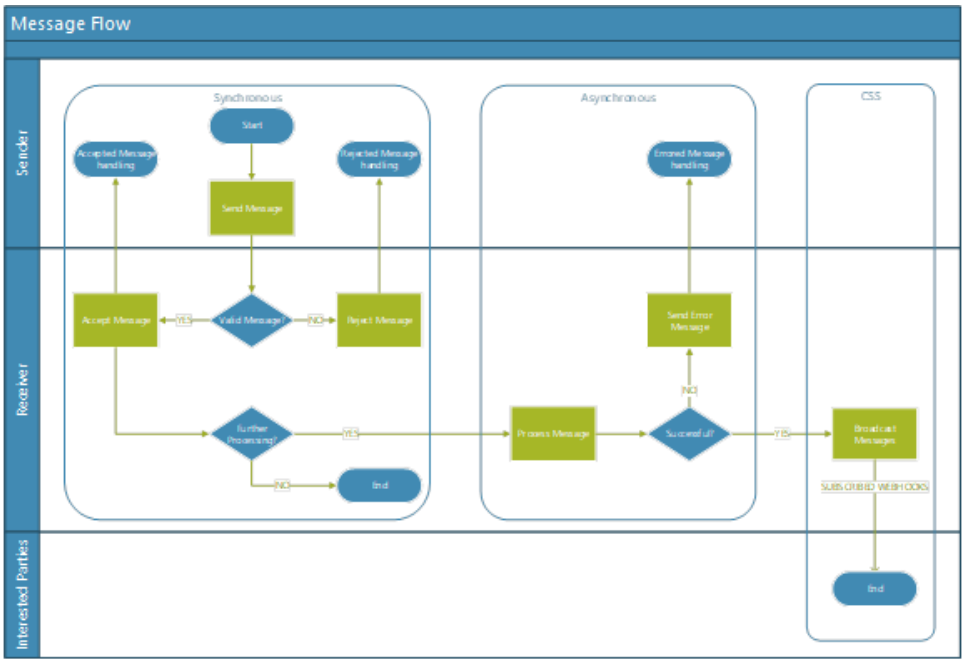
Participants can find information on product specific API routes on the CSS Developer Portal.

Different versions of the API will be available independently in the developer portal and a change log will document any changes made to the new version.

DCC Controlled

7.5.10 Generic Asynchronous Message Flow

The diagram below depicts a generic message flow sequence applicable to both the CSS and interfacing systems. The exception being the CSS Broadcast message shown represents a CSS specific post-processing step, to illustrate how the CSS will inform Market Participants.



The key process steps are:

Send Message
This involves posting a message to a predefined endpoint (URL).

Accept Message
On Receipt of the message a synchronous response is sent to the sender accepting the message (202).

Reject Message
A message is rejected based on schema validation (structure/type/content). The sender remains responsible for resolving the rejected request.

Process Message
The receiver has accepted the message and performs their internal processing of the message content. If this is successful there is no requirement to inform the sender.

Send Error Message
If the internal processing of the message content fails for reasons directly related to the message itself the sender is informed by sending an error message. This message is sent asynchronously.

Message Generation (CSS specific)
This is an illustrative CSS specific post processing step to show how CSS will broadcast information messages to subscribers in response to an incoming message. Other systems may have alternative post processing steps at this point.

7.6 CSS API specification

7.6.1 General Principles

All data passed to CSS should already have had any padding removed.

DCC Controlled

7.6.2 POST

Use POST when you want to create a new resource or initiate a process.

7.6.3 PATCH

Use PATCH when you want to update a subset of a resource.

When sending a PATCH, the fields you wish to “Update” should be sent in the “data” element. You only need to add the properties in the data element that you wish to update, you do not need to send the whole entity. Some properties in the data element may have a dependency on your Role within the CSS.

If a data item is not supplied, it will not be updated.

data:

```
{
        "item1": "item1value"
}
```

If an object in the system looks like the example above:

- Item 1 is an item that just has a value
    - To set item1 to not have a value, send "item1": null
    - To update item1 to have a new value send "item1": "new value"

Use PATCH when you want to delete a resource on a /delete route.

7.6.4 PUT

Use PUT when you need to replace a resource in its entirety.

Use of the PUT method essentially updates the given resource at the request URI. If the request URI refers to an already existing resource – an update operation will happen, otherwise a create operation should happen if the request URI is a valid resource URI.

7.6.5 CorrelationId and EventId

The correlationId will appear in HTTP headers, standard responses, and will also be sent out as part of the webhook payload for related messages.

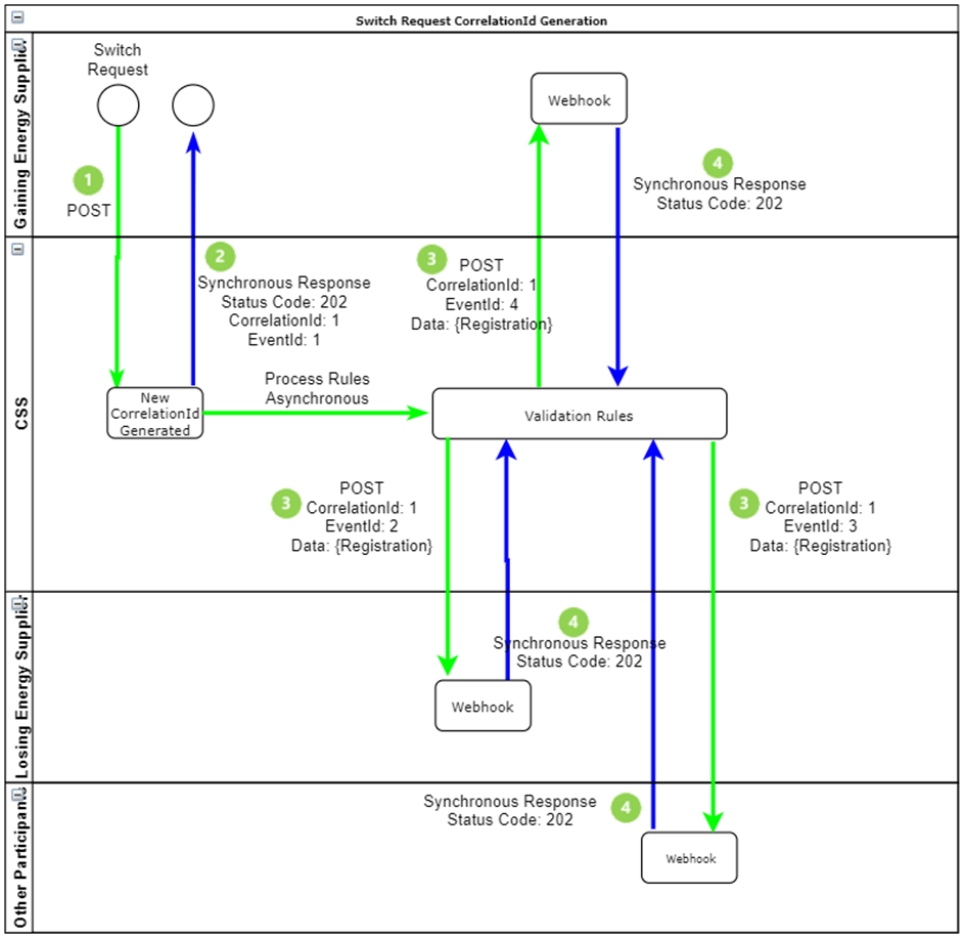
In HTTP headers it will appear in the response field X-Correlation-Id

e.g. X-Correlation-Id: f058ebd6-02f7-4d3f-942e-904344e8cde5

The correlationId can be used by Market Participants to relate messages that are part of the same workflow.

The below is a depiction of how the correlationId and eventId would flow in an asynchronous scenario.

DCC Controlled



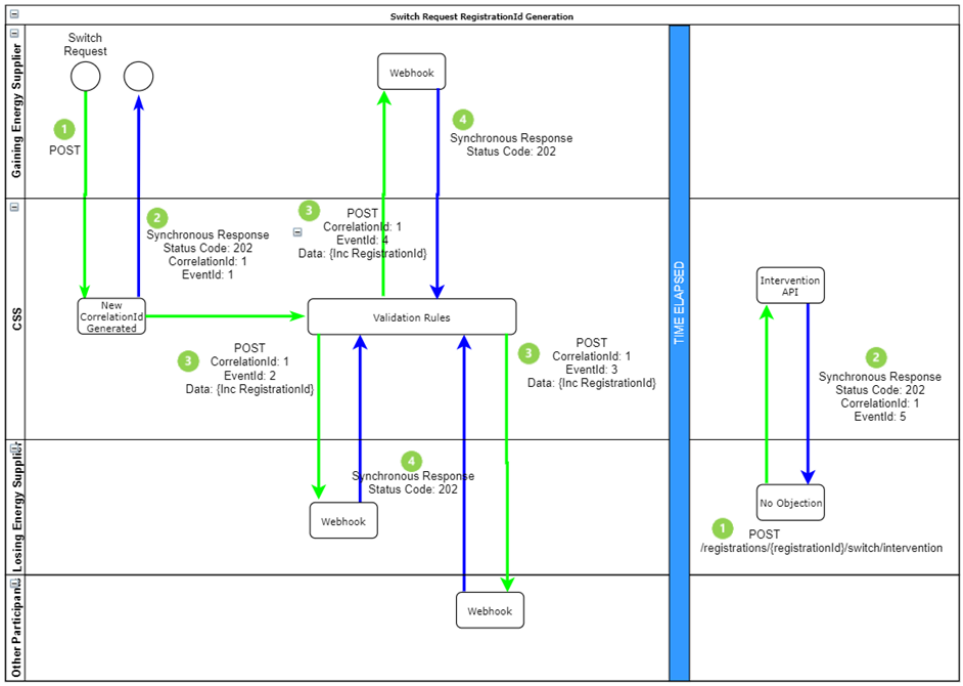
7.6.6 RegistrationId

The CSS will use registration identifiers (registrationId) to uniquely identify each registration. CSS will generate a new registrationId upon receipt of an initial registration or a switch request. All market participants who receive registration lifecycle events will receive the registrationId as part of the Registration Data Payload.

If the losing supplier wishes to object to the proposed switch, they will have to use the newly generated registrationId when intervening. Other market participants will need to store the registrationId for future updates from the CSS and to update the CSS regarding a registration.

The below diagram depicts the basic flow of when and where the registrationId is generated and propagated.

DCC Controlled



7.6.7 Standard Response Body – for information

This is the standard response body sent out synchronously by CSS whenever a message is received by CSS. The errors property will only be populated if the request is in error.

More details are available in the Dev portal

Body:

{

"**version**": "string", --*

"**correlationId**":"UUID", --*

"**eventId**":"UUID", --*

"**eventDate**":"string", --*

"errors": [{Object}]

}

7.7 Error Handling

7.7.1 Synchronous Errors

The goal of error responses is to create a source of information to not only inform the user of a problem, but the possible solution to that problem as well. RFC 7807 provides a standard format for returning problem details from HTTPS APIs, see RFC7807. Whilst we will not follow this to the letter, it does provide some good pointers.

Wrapping an error in an error object gives consumers an easy way to check for an error by simply checking for the existence of the error object. Market Participants should adhere to the format as defined in the CSS Developer Portal when sending synchronous error messages to CSS.

DCC Controlled

7.7.2 Asynchronous Errors

A market participant asynchronous error is where an error occurs in your system as part of your asynchronous processing, after accepting a message from the CSS.

The end point to post the error information to is defined the CSS Developer Portal. The CSS will respond to the asynchronous error message using the response defined in the CSS Developer Portal. These errors will go into the CSS error handling process.

A CSS asynchronous error is where an error occurs in the CSS as part of CSS asynchronous processing. In this scenario the CSS will inform the required market participants of the error, via the market participants Webhook, adhering to the format as defined in the CSS Developer Portal.

7.7.3 Validation

Validation of the Webhook payloads will be controlled via an Open API Specification supplied by CSS on the Dev Portal.

7.7.4 Errors Payload – for illustration

The errors data payload will be sent out to Market Participants as a result of errors within the CSS and also for receiving errors into CSS via the errors API. The http status code for a synchronous response will be the same as the statusCode property within the error object as defined below.

Note: There could be multiple error objects in the "errors" element depending on the event in error.

{

"errors": [{

"**statusCode**": 400,

"**errorCode**": "V1200",

"**errorTitle**": "The switch request is invalid",

"errorDescription": "Property x is invalid, max length is 30 chars", (optional)

"errorType": "https://cssProblems.net/errorCode/V1200" (optional)

}

]

}

Further information on error messages and their payload is available in the Dev portal.

----END OF DOCUMENT---

DCC Controlled