Security Aspects

This section describes in detail what CAP offers to protect your application.

▶ *This guide is available for Node.js and Java.*

Table of Contents

Secure Communications

Encrypted Communication Channels

Integrity and *confidentiality* of data being transferred between any communication endpoints needs to be guaranteed. In particular, this holds true for communication between client and server (public zone resp. platform zone), but also for service-to-service communication (within a platform zone). That means the communication channels are established in a way that rules out undetected data manipulation or disclosure.

Inbound Communication (Server)

SAP BTP exclusively establishes encrypted communication channels based on HTTPS/TLS as shown in the architecture overview and hence fulfills the requirements out of the box. For all deployed (CAP) applications and platform services, the platform's API gateway resp. ingress router provides TLS endpoints accepting incoming request and forwards to the backing services via HTTP. The HTTP endpoints of microservices are only accessible for the router in terms of network technology (perimeter security) and therefore aren't visible for clients in public and platform zone. Likewise microservices can only serve a single network port, which the platform has opened for the hosting container.

The router endpoints are configured with an up to date TLS protocol version containing a state-of-the-art cipher suite. Server authentication is given by X.509 server certificates signed by a trusted certificate authority.

> **TIP**
>
> It's mandatory for public clients to authenticate the server and to verify the server's identity by matching the target host name with the host name in the server certificate.

> **TIP**
>
> Manually provided certificates for custom domains need to be signed by a trusted certificate authority .

Outbound Communication (Client)

As platform services and other applications deployed to BTP are only accessible via exposed TLS router endpoints, outbound connections are automatically secured as well. Consequently, technical clients have to validate the server certificate for proper server authentication. Also here CAP application developers don't need to deal with HTTPS/TLS connection setup provided the client code is build on CAP offerings such as HANA Cloud Service or CloudSDK integration.

> **WARNING**
>
> The **CAP application needs to ensure adequate protection of secrets** that are injected into CAP microservices, for example:
>
> - mTLS authentication is enabled in the XSUAA service instance of your application and also for XSUAA reuse instances of platform services.
> - Ensure that service bindings and keys aren't compromised (rotate regularly).
> - SAP BTP Connectivity services are maintained securely .

Internal Communication (Client and Server)

Depending on the target platform, closely coupled microservices of the application zone might also communicate via trusted network channels instead of using outbound connections. For instance, a CAP service could communicate to a CAP sidecar, which is deployed to the same container via localhost HTTP connection.

> **TIP**
>
> CAP allows to use alternative communication channels, but application operators are responsible to set up them in a secure manner.

> **TIP**
>
> CAP applications don't have to deal with TLS, communication encryption, or certificates, for inbound as well as outbound connections.

Filtering Internet Traffic

Reducing attack surface by filtering communication from or to public zone increases the overall security protection level. By default, the platform comes with a standard set of services and configurations to protect network communication building on security features of the underlying hyperscaler.

> **WARNING**
>
> Measures to further **restrict web access to your application** can be applied at platform level and aren't offered by CAP. For instance, CF Route service can be used to implement route-specific restriction rules.

Secure Authentication

None-public resources may only be accessed by authenticated users. Hence, authentication plays a key role for product security on different levels:

- **Business users** consume the application via web interface. In multitenant applications, they come from different subscriber tenants that need to be isolated from each other.
- **Platform users** operate the application and have privileged access to its components on OS level (containers, configurations, logs etc.). Platform users come from the provider tenant.

Managing user pools, providing a logon flow, and processing authentication are complex and highly security-critical tasks **that shouldn't be tackled by applications**. Instead, applications should rely on an identity service provided by the platform which is seamlessly integrated by CAP.

Find more about platform and business users:
↳ *SAP BTP User and Member Management*

Server Requests

SAP BTP offers central identity services SAP Cloud Identity Services - Identity Authentication resp. SAP Authorization and Trust Management Service for managing

and authenticating platform and business users providing:

- User authentication flows (OpenID connect), for example, multifactor authentication
- Federation of custom identity providers (IdPs)
- Single-sign on
- Principal propagation
- Password and session policies etc.

The central platform service provides applications with a large set of industry-proven security features, which is why applications don't have to develop their own extensions and run the risk of security flaws.

CAP doesn't require any specific authentication strategy, but it provides out of the box integration with the platform identity service. On configured authentication, *all CAP endpoints are authenticated by default*.

> **WARNING**
>
> ❗ **CAP applications need to ensure that an appropriate authentication method is configured**. It's highly recommended to establish integration tests to safeguard a valid configuration.

Learn more about user model and identity providers here:
↳ *SAP BTP Security*

Remote Services

CAP microservices consume remote services and hence need to be authenticated as technical client as well. Similar to request authentication, CAP saves applications from having to implement secure setup of service to service communication:

- CAP interacts with platform services such as Event Mesh or SaaS Provisioning Service on basis of platform-injected service bindings.
- CAP offers consumption of Remote Services on basis of SAP BTP destinations.

Note that the applied authentication strategy is specified by server offering and resp. configuration and not limited by CAP.

Maintaining Sessions

CAP microservices require authentication of all requests, but they don't support logon flows for UI clients. Being stateless, they neither establish a session with the client to store login information such as an OAuth 2 token that needs to be passed in each server request.

To close this gap, UI-based CAP applications can use an Application Router instance or service as reverse proxy as depicted in the diagram. The Application Router redirects the login to the identity service, fetches an OAuth2 token, and stores it into a secure session cookie.

> **WARNING**
>
> ❗ The **Application Router endpoints don't hide CAP endpoints** in the service backend. Hence, authentication is still mandatory for CAP microservices.

Maintaining Secrets

To run a CAP application that authenticates users and consumes remote services, **it isn't required to manage any secrets such as keys, tokens, or passwords**. Also CAP doesn't store any of them, but relies on platform injection mechanisms or destinations.

> **TIP**
>
> In case you still need to store any secrets, use a platform service SAP Credential Store .

Secure Authorization

According to segregation of duties paradigm, user administrators need to control how different users may interact with the application. Critical combinations of authorizations must be avoided. Basically, access rules for business users are different from platform users.

Business Users

To align with the principle of least privilege, applications need to enforce fine-grained access control for business users from the subscriber tenants.

Depending from the business scenario, users need to be restricted to operations they perform on server resources, for example, reading an entity collection. Moreover, they might also be limited to a subset of data entries, that is, they may only operate on a filtered view on the data. The set of rules that apply to a user reflects a specific conceptual role that describes the interaction with the application to fulfill a business scenario. Obviously, the business roles are dependent from the scenarios and hence *need to be defined by the application developers*.

Enforcing authorization rules at runtime is highly security-critical and shouldn't be implemented by the application as this would introduce the risk of security flaws. Instead, CAP authorizations follow a declarative approach allowing applications to design comprehensive access rules in the CDS model.

Resources in the model such as services or entities can be restricted to users that fulfill specific conditions as declared in `@requires` or `@restrict` annotations. According to the declarations, server-side authorization enforcement is guaranteed for all requests. It's executed close before accessing the corresponding resources.

> **WARNING**
>
> ❗ **By default, CAP services and entities aren't authorized**. Application developers need to **design and test access rules** according to the business need.

> **TIP**
>
> To verify CAP authorizations in your model, it's recommended to use CDS lint rules.

The rules prepared by application developers are applied to business users according to grants given by the subscribers user administrator, that is, they're applied tenant-specific.

CAP authorizations can be defined dependently from user claims such as XSUAA scopes or attributes that are deployed by application developers and granted by the user administrator of the subscriber. Hence, CAP provides a seamless integration of central identity service without technical lock-in.

> **TIP**

> You can generate the `xs-security.json` descriptor file of the application's XSUAA instance by executing `cds add xsuaa` in the project root folder. The XSUAA scopes, roles, and attributes are derived from the CAP authorization model.

> **WARNING**
>
> CAP authorization enforcement doesn't automatically log successful and unsuccessful authorization checks. Applications need to add corresponding custom handlers to support it.

Authorization of CAP Endpoints

In general, responses created by *standard* CAP handlers and services are created on need-to-know basis. This means, authorized users only receive server information according to their privilege. Therefore, business users won't gain information about server host names, any version of application server component, generated queries etc.

Based on the CDS model and configuration of CDS services, the CAP runtime exposes following endpoints:

Name	Configuration	URL	Authorization
CDS Service `Foo`	`service Foo {}`	`/<protocol-path>/Foo/**` [1]	`@restrict` / `@requires` [2]
	OData v2/v4	`/<odata-path>/Foo/$metadata` [1]	See here
Index page		`/index.html`	none, but disabled in production

> [1] See protocols and paths

> [2] No authorization by default

Based on configured features, the CAP runtime exposes additional callback endpoints for specific platform service:

Platform service	URL	Authorization
Multitenancy (SaaS Registry)	none so far for Node.js	

Moreover, technical MTXs CAP services may be configured, for example, as sidecar microservice to support higher-level features such as Feature Toggles or Multitenancy:

CAP service	URL	Authorization
cds.xt.ModelProviderService	`/-/cds/model-provider/**`	Internal, technical user[1]
cds.xt.DeploymentService	`/-/cds/deployment/**`	
cds.xt.SaasProvisioningService	`/-/cds/saas-provisioning/**`	Internal, technical user[1], or technical roles `cds.Subscriber` resp. `mtcallback`
cds.xt.ExtensibilityService	`/-/cds/extensibility/**`	Internal, technical user[1], or technical roles `cds.ExtensionDeveloper`

[1] The microservice running the MTXS CAP service needs to be deployed to the application zone and hence has established trust with the CAP application client, for instance given by shared XSUAA instance.

Authentication for a CAP sidecar needs to be configured just like any other CAP application.

> **WARNING**
>
> ❗ Ensure that technical roles such as `cds.Subscriber`, `mtcallback`, or `emcallback` **are never included in business roles**.

Platform Users

Similar to business consumption, different scenarios apply on operator level that need to be separated by dedicated access rules: deployment resp. configuration, monitoring, support, audit logs etc. *CAP doesn't cover authorization of platform users*. Please refer to security documentation of the underlying SAP BTP runtime environment:

- Roles in the Cloud Foundry Environment
- Roles in the Kyma Environment

Secure Multi-Tenancy

Multitenant SaaS-applications need to take care for security aspects on a higher level. Different subscriber tenants share the same runtime stack to interact with the CAP application. Ideally, from perspective of a single tenant, the runtime should look like a self-contained virtual system that doesn't interfere with any other tenant.

All directly or indirectly involved services that process the business request require to isolate with regards to several dimensions:

- No breakout to persisted data
- No breakout to transient data
- Limited resource consumption

The CAP runtime is designed from scratch to support tenant isolation:

Isolated Persistent Data

Having configured Multitenancy in CAP, when serving a business request, CAP automatically targets an isolated HDI container dedicated for the request tenant to execute DB statements. Here, CAP's data query API based on CQN is orthogonal to multitenancy, that is, custom CAP handlers can be implemented agnostic to MT.

During tenant onboarding process, CAP triggers the HDI container creation via SAP HANA Cloud Services . The containers have separated DB schemas and dedicated technical DB users for access. CAP guarantees that code for business requests runs on a DB connection opened for the technical user of the tenant's container.

Isolated Transient Data

Although CAP microservices are stateless, the CAP Java runtime (generic handlers inclusive) needs to cache data in-memory for performance reasons. For instance, filters for instance-based authorization are constructed only once and are reused in subsequent requests.

Request-related data is propagated down the call stack via the continuation-local variable cds.context.

> **WARNING**
>
> Make sure that custom code doesn't break tenant data isolation or leak data across concurrent requests.

As a best practice, you should not put any non-static variables in the closures of your service implementations.

Bad example:

srv/cat-service.js

```js
module.exports = srv => {
  let books  // <- leaks data across tenants and concurrent requests
  srv.on('READ', 'Books', async function(req, next) {
    if (books) return books
    return books = await next()
  })
}
```

Limiting Resource Consumption

Tenant-aware microservices also need to handle resource consumption of tenants, in particular with regards to CPU, memory, and network connections. Excessive use of resources requested by a single tenant could cause runtime problems for other consumers (noisy neighbor problem).

CAP helps to control resource usage:

- Fine granular processing of request (CAP handlers) to avoid disproportionate blocking times of the event loop.
- Tenants have dedicated DB connection pools.

> **TIP**
>
> Make sure that custom code doesn't introduce excessive memory or CPU consumption within a single request.

Because OS resources are strictly limited in a virtualized environment, a single microservice instance can handle load of a limited set of tenants, only. **Adequate sizing** of your microservice is mandatory, that is, adjusting memory settings, connection pool sizes, request size limits etc. according to the business needs.

Last but not least you need to implement a **scaling strategy** to meet increasing load requirements by additional microservice instances.

> **WARNING**
>
> ❗ **Sizing and scaling** is up to application developers and operators. CAP default values aren't suitable for all applications.

Secure Against Untrusted Input

Without protection mechanism in place, a malicious user could misuse a valid (that is, authenticated) session with the server and attack valuable business assets.

Injection Attacks

Attackers can send malicious input data in a regular request to make the server perform unintended actions that can lead to serious data exploits.

Common Attack Patterns

- CAP's intrinsic data querying engine is immune with regards to SQL injections that are introduced by query parameter values that are derived from malicious user input. CQL statements are transformed into prepared statements that are executed in SQL databases such as SAP HANA. Be aware that injections are still possible even via CQL when the query structure (target entity, columns and so on) is based on user input:

```js
const entity = <from user input>
const column = <from user input>
validate(entity, column) // for example, by comparing with positive list
SELECT.from(entity).columns(column)
```

> **WARNING**
>
> Be careful with custom code when creating or modifying CQL queries. Additional input validation is needed when the query structure depends on the request's input.

- Cross Site Scripting (XSS) is used by attackers to inject a malicious script, which is executed in the browser session of an unsuspecting user. By default, there are some protection mechanisms in place. For instance, CAP OData V4 adapter renders responses with HTTP, which prevents the browser from misinterpreting the context. On the client side, SAPUI5 provides input validation for all typed element properties and automatic output encoding in all standard controls.

- Untrusted data being transferred may contain malware. SAP Malware Scanning Service is capable to scan provided input streams for viruses and is regularly updated.

  > **WARNING**
  >
  > ❗ Currently, CAP applications need to add custom handlers to **scan data being uploaded or downloaded**.

- Path traversal attacks aim to access parts of the server's file system outside the web root folder. As part of the application zone, an Application Router serves the static UI content of the application. The CAP microservice doesn't need to serve web content from file system. Apart from that the used web server frameworks such as Spring or Express already have adequate protection mechanisms in place.

- CLRF injections or log injections can occur when untrusted user input is written to log output.

  CAP Node.js offers a CLRF-safe logging API that should be used for application logs.

- Deserialization of untrusted data can lead to serious exploits including remote code execution. The OData adapter converts JSON payload into an object representation. Here it follows a hardened deserialization process where the deserializer capabilities (for example, no default types in Jackson) are restricted to a minimum. A strong input validation based on EDMX model is done as well. Moreover, deserialization errors terminate the request and are tracked in the application log.

General Recommendations Against Injections

In general, to achieve perfect injection resistance, applications should have input validation, output validation, and a proper Content-Security-Policy in place.

- CAP provides built-in support for **input validation**. Developers can use the `@assert` annotation to define field-specific input checks.

> **WARNING**
>
> Applications need to validate or sanitize all input variables according to the business context.

- With respect to **output encoding**, CAP OData adapters have proper URI encoding for all resource locations in place. Moreover, OData validates the JSON response according to the given EDMX schema. In addition, client-side protection is given by SAPUI5 standard controls
- Applications should meet basic Content Security Policy (CSP) compliance rules to further limit the attack vector on client side. CSP-compatible browsers only load resources from web locations that are listed in the allowlist defined by the server. `Content-Security-Policy` header can be set as route-specific response header in the Application Router . SAPUI5 is CSP-compliant as well.

> **WARNING**
>
> Applications have to **configure Content Security Policy** to meet basic compliance.

Service Misuse Attacks

- Server Side Request Forgery (SSRF) abuses server functionality to read or update resources from a secondary system. CAP microservices are protected from this kind of attack if they use the CAP standard mechanisms for service to service communication.
- Cross-Site Request Forgery (CSRF) attacks make end users executing unwanted actions on the server while having established a valid web session. By default, the Application Router, which manages the session with the client, enforces a CSRF token protection (on basis of `x-csrf-token` headers). Hence, CAP services don't have to deal with CSRF protection as long as they don't maintain sessions with the client. SAPUI5 supports CSRF tokens on client side out of the box.
- Clickjacking is an attack on client side where end users are tricked to open foreign pages. SAPUI5 provides protection mechanisms against this kind of attack.

> **WARNING**
>
> To protect SAPUI5 applications against clickjacking, configure `frame options` .

Denial-of-Service Attacks

Denial-of-service (DoS) attacks attempt to reduce service availability for legitimate users. This can happen by erroneous server behavior upon a single large or a few specially crafted malicious requests that bind an excessive amount of shared OS resources such as CPU, memory, or network connections.

Since OS resource allocations are distributed over the entire request, DoS-prevention needs to be addressed in all different layers of the runtime stack:

HTTP Server and CAP Protocol Adapter

The used web server frameworks such as Spring/Tomcat or Express start with reasonable default limits, for example:

- Maximum size of the HTTP request header.
- Maximum size of the HTTP request body.
- Maximum queue length for incoming connection requests.
- Maximum number of connections that the server accepts and processes at any given time.
- Connection timeout. Additional size limits and timeouts (request timeout) are established by the reverse proxy components, API Gateway and Application Router.

> **TIP**
>
> If you want to apply an application-specific sizing, consult the corresponding framework documentation.
>
> See section Maximum Request Body Size to find out how to restrict incoming requests to a CAP Node.js application depending on the body size.

Moreover, CAP adapters automatically introduce query results pagination in order to limit memory peaks (customize with `@cds.query.limit`). The total number of request of OData batches can be limited by application configuration.

> **WARNING**
>
> ❗ CAP applications have to limit the amount of `$expands` per request in a custom handler. Also, the maximum amount of requests per `$batch` request need to be configured with `cds.odata.batch_limit = <max_requests>` ☼

> **TIP**

> Design your CDS services exposed to web adapters on need-to-know basis. Be especially careful when exposing associations.

CAP Service Runtime

Open transactions are expensive as they bind many resources such as a database connection as well as memory buffers. To minimize the amount of time a transaction must be kept open, the CAP runtime offers an Outbox Service that allows to schedule asynchronous remote calls in the business transaction. Hence, the request time to process a business query, which requires a remote call (such as to an audit log server or messaging broker), is minimized and independent from the response time of the remote service.

> **TIP**
>
> Avoid synchronous requests to remote systems during a transaction.

↳ *See why CPU time is fairly distributed among business requests*

Database

As already outlined, database connections are a expensive resource. To limit overall usage, by default, the CAP runtime creates connection pools per subscriber tenant. Similarly, the DB driver settings such as SQL query timeout and buffer size have reasonable values.

> **TIP**
> In case the default setting doesn't fit, connection pool properties and driver settings can be customized, respectively.

> **WARNING**
>
> ❗ Applications need to establish an adequate Workload Management that controls DB resource usage.

Supplementary Measures

As outlined before, a well-sized microservice instance doesn't help to protect from service downtimes when excessive workload initiated by an attacker exceeds the available capacity. Rate limiting is a possible counter measure to restrict the frequency of calls of a client.

> **WARNING**
>
> ❗ Applications need to establish an adequate **rate limiting** strategy.

There's also the possibility to introduce request filtering and rate limiting on platform level via Route Service . It has the advantage that the requests can be controlled centrally before touching application service instances.

In addition, the number of instances need to be **scaled horizontally** according to current load requirements. This can be achieved automatically by consuming Application Autoscaler .

Additional Protection Mechanisms

There are additional attack vectors to consider. For instance, naive URL handling in the server endpoints frequently introduces security gaps. Luckily, CAP applications don't have to implement HTTP/URL processing on their own as CAP offers sophisticated protocol adapters such as OData V2/V4 that have the necessary security validations in place. The adapters also transform the HTTP requests into a corresponding CQN statement. Access control is performed on basis of CQN level according to the CDS model and hence HTTP Verb Tampering attacks are avoided. Also HTTP method override, using `X-Http-Method-Override` or `X-Http-Method` header, is not accepted by the runtime.

The OData protocol allows to encode field values in query parameters of the request URL or in the response headers. This is, for example, used to specify:

- Pagination (implicit sort order)
- Searching Data
- Filtering

> **WARNING**
>
> Applications need to ensure by means of CDS modeling that fields reflecting sensitive data are excluded and don't appear in URLs.

> **TIP**
>
> It's recommended to serve all application endpoints via CAP adapters. Securing custom endpoints is left to the application.

In addition, CAP runs on a virtual machine with a managed heap that protects from common memory corruption vulnerabilities such as buffer overflow or range overflows.

CAP also brings some tools to effectively reduce the attack vector of race condition vulnerabilities. These might be exposed when the state of resources can be manipulated concurrently and a consumer faces an unexpected state. CAP provides basic means of concurrency control on different layers, for example ETags and pessimistic locks. Moreover, Messages received from the message queue are always in order.

> **TIP**
>
> Applications have to ensure a consistent data processing taking concurrency into account.

Secure by Default and by Design

Secure Default Configuration

Where possible, CAP default configuration matches the secure by default principle:

- There's no need to provide any password, credentials, or certificates to protect communication.
- A CAP application bound to an XSUAA instance authenticates all endpoints by default. Developers have to explicitly configure public endpoints if necessary.
- Isolated multitenancy is provided out of the box.
- Application logging has `INFO` level to avoid potential information disclosures.
- CAP also has first-class citizen support for Fiori UI framework that brings a lot secure by default features in the UI client.

Of course, several security aspects need application-specific configuration. For instance, this is true for authorizations or application sizing.

> **TIP**
>
> It's recommended to ensure security settings by automated integration tests.

CAP provides some features that are suitable for development only such as

- Index Page
- Mock Users
- Developer Dashboard (Java only)

These features are deactivated in the production profile by default.

> **WARNING**
>
> **Do not manually enable features for production that are disabled by the production profile**, as this could introduce serious security vulnerabilities.

↳ *Learn more about production profiles in Java*

↳ *Learn more about production profiles in Node.js*

Fail Securely

CAP runtime differentiates several types of error situations during request processing:

- Exceptions because of invalid user input (HTTP 4xx).
- Exceptions because of unexpected server behaviour, for example, network issues.
- Unrecoverable errors due to serious issues in the VM (for example, lack of memory) or program flaws.

In general, **exceptions immediately stop the execution of the current request**.

CAP Node.js adds an exception wrapper to ensure that only the failing request is affected by the exception.

Customers can react in dedicated exception handlers if necessary.

In contrast, **errors stop the overall microservice** to ensure that security measures aren't weakened.

> **TIP**
>
> Align the exception handling in your custom coding with the provided exception handling capabilities of the CAP runtime.

Edit this page

Last updated: 29/08/2025, 09:36

Previous page
Platform Security

Next page
Data Protection & Privacy

Was this page helpful?