CalConnect TC VCARD

Common architecture for portable secure information interchange and unified management (Capsium)

Working Draft Standard

Warning for drafts

This document is not a CalConnect Standard. It is distributed for review and comment, and is subject to change without notice and may not be referred to as a Standard. Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

The Calendaring and Scheduling Consortium, Inc. 2024

The Calendaring and Scheduling Consortium, Inc.

4390 Chaffin Lane
McKinleyville
California 95519
United States of America

copyright@calconnect.org
www.calconnect.org

Contents

Abstract

Capsium is a modular framework designed to efficiently and securely interchange multi-format information in interoperable portable packages, as well as platform-independent deployment of these packages to serve information consumers.

This document specifies requirements of the Capsium framework and its components:

— Capsium packages: standardized unit of information interchange in the Capsium framework.
— Capsium reactors: server-side or user-side software that allows deployment of a Capsium package.
— Capsium HTTP API: user-facing HTTP API implemented by a Capsium reactor.

Introduction

General

The digital era demands advanced, secure, and efficient methods for deploying and exchanging information. As organizations contend with multi-format data across diverse platforms, the limitations of traditional web packaging solutions become increasingly apparent.

Capsium answers this need with a modular framework designed for the efficient and secure interchange of multi-format information within interoperable and portable packages.

Capsium uniquely supports the packaging and deployment of “non-application websites” or “non-server-side application websites” by delegating those functions to the Capsium reactor, which does not depend on a web server and can be directly implemented by the browser, mimicking the file-serving capabilities of a web server.

Features

Today’s complex digital landscape highlight the necessity for Capsium.

— Portability: There is a critical need for data and applications to be easily transferable across different environments without compromising security or functionality. Capsium ensures that packages can be seamlessly moved between platforms.
— Data immutability: Ensuring that data remains unchanged and secure from tampering is essential for maintaining integrity and trust. Capsium guarantees data immutability through advanced cryptographic techniques, crucial for compliance and auditing.
— Interoperability: Diverse systems and applications must communicate effectively. Capsium supports a wide range of formats and protocols, ensuring seamless integration and data exchange across different platforms.
— Single-page applications (SPAs): Modern web applications demand dynamic, responsive user experiences. Capsium supports the development and deployment of SPAs, reducing server dependency and enhancing performance.
— Capsium Reactors: To meet varied deployment needs, Capsium introduces reactors that can be installed on user machines or servers, managing and deploying Capsium packages with flexibility and scalability, without the need for a traditional web server.

Comparison with existing solutions

Capsium’s unique approach addresses the deficiencies of existing packaging solutions for non-application websites and web applications, which are all unsuitable for the use case.

The following packaging solutions are listed in order of an decreasing level of virtualization.

— Website bundles:
  — Examples: Safari webarchive (extension `.webarchive`), WARC (extension: `.warc`), Mozilla Archive Format (MAFF, `.maff`), Microsoft Compiled HTML Help (CHM, extension `.chm`) and MHTML/MHT (MIME Encapsulation of Aggregate HTML Documents, extension: `.mhtml`, `.mht`) (defined in RFC 2110 and RFC 2557).
  — Purpose: Bundle website resources for offline access and archival.
  — Limitations: Not interoperable and difficult to implement across different browsers. Do not support server-side deployments. Unable to contain single-page applications that require rich HTML API interfaces.

— Client-side web application bundles:
  — Examples: Electron, NW.js.
  — Purpose: Bundle a browser and a web server along with necessary language interpreters.
  — Limitations: Resulting bundles are often very large and cumbersome to distribute, requiring significant memory for simple outputs. Introduce non-platform-independent executable code, complicating data management processes.

— Server-side web application bundles:

  - Examples: Webpack, Gulp (JavaScript); Maven (Java); Bundler (Ruby); pip and setuptools (Python).
  - Purpose: Automate bundling of static assets and dependencies, improving load times and simplifying development workflows.
  - Limitations: Require a large number of dependencies and permissions for port opening and listening. Necessitate running a server, which can be complex and resource-intensive.

- Containers:
  - Examples: Docker, LXC.
  - Purpose: Allow entire applications and their dependencies to be packaged into portable containers, enhancing deployment consistency.
  - Limitations: Require virtualization permissions on the machine, which can be a barrier for some environments. Resource-intensive.

- Virtual machines:
  - Examples: VMWare, Xen.
  - Purpose: Provide complete isolation and can run different operating systems on a single hardware host.
  - Limitations: Require permissions at the hardware level and can be resource-intensive.

Benefits

Capsium addresses the shortcomings of previous solutions and offers unique benefits:

- Interoperability: Supports a variety of formats and protocols, enabling seamless communication between different systems and platforms.
- Portability: Capsium packages are easily transferable, facilitating data migration and deployment without compromising security or functionality.
- Efficiency: Optimizes deployment processes for SPAs and static websites, reducing server dependency and improving performance.
- Security: Utilizing advanced encryption and key management, Capsium ensures secure storage and transfer of information.
- Compliance and auditing: Features for tracking data access and modifications ensure regulatory compliance and robust auditing capabilities.
- Capsium reactors: Provide flexible and scalable package management and deployment solutions, eliminating the need for traditional server infrastructure.

Data management impact

Capsium modernizes and transforms data management practices in several key ways:

- Enhanced Security: Prioritizes data immutability and advanced encryption, helping organizations mitigate data breach risks.
- Improved Interoperability: Facilitates greater integration and communication across different systems, driving innovation and efficiency.
- Portability: Simplifies the transfer of data and applications across different environments, reducing the effort and risk associated with migration.
- Efficiency: Streamlines deployment processes, particularly for SPAs and static websites, leading to faster load times and reduced server loads.
- Compliance and Auditing: Facilitates adherence to regulatory requirements by ensuring data integrity and providing robust tracking of data access and modifications.

Summary

Capsium represents a significant advancement in the realm of data management, addressing critical needs for security, interoperability, portability, efficiency, and compliance. Its innovative approach transforms how organizations handle data, making it a vital tool in the modern digital landscape.

Validate composite package

capsium-validate /composite-package `

Security

General

Security is a critical aspect of the Capsium package, ensuring data integrity and protection. This section describes the requirements and specifications for implementing security features, with detailed use cases illustrating practical applications.

Integrity checks	Mechanisms to ensure that the package's content has not been tampered with.
Sandboxing packaged application	Isolates the execution of the package's content to prevent security breaches.

Digital signatures are an optional capability defined by the signatures module ([module-signatures]).

Example:

```
{
    "security": {
        "integrityChecks": {
            "checksumAlgorithm": "SHA-256",
            "checksums": {
                "index.html": "d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2
d2d2d2d2d2d2d2d2",
                "styles.css": "e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3
e3e3e3e3e3e3e3e3"
            }
        },
        "sandboxing": {
            "enabled": true,
            "parameters": {
                "memoryLimit": "512MB",
                "cpuLimit": "2"
            }
        }
    }
}
```

Figure 68

Integrity Checks

Requirements and Specifications

1) Checksum Calculation:

checksumAlgorithm — The algorithm used to calculate checksums for the package's content.

Type	`string`
Values	`"SHA-256"`
Example	`"SHA-256"`

checksums — A dictionary where the keys are file paths and the values are their respective checksums.

Type	`object`

Format	`{ "filePath": "checksumValue" }`
Example	

```
{
    "index.html": "d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2",
    "styles.css": "e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3"
}
```

Figure 69

1) Integrity Verification:
   — The system must verify the integrity of the package's content by comparing the calculated checksums with those in the manifest file.
   — Any discrepancies must trigger an alert or rejection of the package.

Use Case	Content Integrity	After downloading the Capsium package, the system calculates checksums for all files and compares them to the provided checksums to ensure no files were altered during transit.
Example Configuration		

```
{
    "security": {
        "integrityChecks": {
            "checksumAlgorithm": "SHA-256",
            "checksums": {
                "index.html": "d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2",
                "styles.css": "e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3"
            }
        }
    }
}
```

Figure 70

Procedure to Calculate the Integrity Hash of a Capsium Package	1) Select the Checksum Algorithm: — Choose an algorithm from the supported options (e.g., SHA-256). — Example: `checksumAlgorithm: "SHA-256"` 2) Calculate Checksums: — For each file in the package, calculate the checksum using the selected algorithm. — Use a reliable tool or library to perform the checksum calculation. — Example command using `sha256sum`:

```
sha256sum index.html > checksums.txt
sha256sum styles.css >> checksums.txt
```

Figure 71

1) Create a Checksums Manifest:
   — Compile the calculated checksums into a JSON object.
   — Ensure the file paths are correctly mapped to their respective checksums.
   — Example:


```
{
        "index.html": "d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2",
        "styles.css": "e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3"
    }
```

Figure 72

1) Store the Checksum Manifest:
   - Save the checksum manifest file (e.g., `checksums.json`) within the package.
   - Ensure this file is included when distributing the package.
2) Verify the Integrity:
   - Upon receiving the package, recalculate the checksums for each file using the same algorithm.
   - Compare the newly calculated checksums with those in the `checksums.json` manifest.
   - If all checksums match, the package integrity is verified. If any checksum does not match, reject the package as it may have been tampered with.

Sandboxing Packaged Applications

Requirements and Specifications

1) Sandboxing Environment:
   - enabled: Indicates whether sandboxing is enabled.
     - Type: `boolean`
     - Values: `true`, `false`
     - Example: `true`
   - parameters: An object specifying resource limits for the sandbox.
     - Type: `object`
     - Attributes:
       - memoryLimit: The maximum amount of memory allocated to the sandbox.
         - Type: `string`
         - Example: `"512MB"`
       - cpuLimit: The maximum number of CPU cores allocated to the sandbox.
         - Type: `number`
         - Example: `2`
2) Isolation Mechanism:
   - The sandbox should prevent the package from accessing or modifying system resources outside its designated environment.
   - The sandbox should enforce strict boundaries to minimize the risk of security breaches.

Use Case

Running Untrusted Code: When deploying a Capsium package that contains untrusted or third-party code, the sandbox ensures that the code runs in isolation, preventing it from affecting the host system.

Example Configuration

```
{
    "security": {
        "sandboxing": {
            "enabled": true,
            "parameters": {
                "memoryLimit": "512MB",
                "cpuLimit": 2
            }
        }
    }
}
```

Figure 73

By implementing these security features, including a detailed procedure for calculating and verifying integrity hashes, the Capsium package ensures high standards of data integrity and protection, safeguarding both the package content and the systems it interacts with.

Encrypted content is an optional capability defined by the encryption module ([module-encryption]).

Validation

General

Validation ensures the quality and correctness of the Capsium package. This section describes the various quality and correctness checks that can be performed on a Capsium package, along with their attributes and features.

Data validation

Data sets included should be validated against respective schemas to be correct otherwise operational deployment of the Capsium package will fail.

Quality Checks

Quality checks are processes designed to verify the quality of the package's content. These checks ensure that the package adheres to best practices and standards for code quality and content integrity.

HTML Validation	Ensures that all HTML files in the package are well-formed and comply with HTML standards.
CSS Validation	Checks that all CSS files are syntactically correct and conform to CSS specifications.
JavaScript Linting	Uses a linter tool to analyze JavaScript code for potential errors and adherence to coding standards.

Example:

```
{
    "validation": {
        "qualityChecks": {
            "htmlValidation": true,
            "cssValidation": true,
            "jsLinting": true
```



```
        }
    }
}
```

Figure 74

In this example, all three quality checks (`htmlValidation`, `cssValidation`, and `jsLinting`) are enabled, indicating that HTML, CSS, and JavaScript files will be validated.

Correctness Checks

Correctness checks ensure that the package meets all specifications and requirements. These checks validate the structural and functional integrity of the package.

Schema Validation	Ensures that data files within the package conform to predefined schemas. This is critical for maintaining consistency and correctness in data formats.
Dependency Validation	Checks that all dependencies required by the package are correctly specified and available. This ensures that the package can be built and run without missing dependencies.

Example:

```
{
    "validation": {
        "correctnessChecks": {
            "schemaValidation": true,
            "dependencyValidation": true
        }
    }
}
```

Figure 75

In this example, both `schemaValidation` and `dependencyValidation` are enabled, indicating that the package's data files will be validated against their schemas, and all dependencies will be checked for correctness.

Combined Example

A comprehensive validation configuration that includes both quality and correctness checks might look like this:

```
{
    "validation": {
        "qualityChecks": {
            "htmlValidation": true,
            "cssValidation": true,
            "jsLinting": true
        },
        "correctnessChecks": {
            "schemaValidation": true,
            "dependencyValidation": true
        }
    }
}
```

Figure 76

In this expanded example, the validation object contains both quality checks and correctness checks, providing a holistic validation approach to ensure the package is both high-quality and correct.

Attributes Summary

validation The root object for validation configuration.

- qualityChecks Object containing quality check configurations.
  - htmlValidation (boolean): Enable or disable HTML validation.
  - cssValidation (boolean): Enable or disable CSS validation.
  - jsLinting (boolean): Enable or disable JavaScript linting.
- correctnessChecks Object containing correctness check configurations.
  - schemaValidation (boolean): Enable or disable schema validation.
  - dependencyValidation (boolean): Enable or disable dependency validation.

By configuring these attributes, Capsium packages can be thoroughly validated to ensure they meet both quality and correctness standards. This structured approach helps maintain high standards and reliability for Capsium packages.

Packaging options

General

Encapsulation in the context of Capsium packages involves ensuring data integrity, security, and efficiency through compression. These operations are configured and managed through a `packaging.json` file.

The `packaging.json` file serves as a centralized configuration for managing the compression of the Capsium package encapsulation process. It ensures that all necessary parameters and standards are clearly defined and easily accessible for implementation.

Compression is always applied. Digital signing and encryption of the package are optional capabilities defined by the signatures and encryption modules ([module-signatures], [module-encryption]); when claimed, their configuration is likewise recorded in `packaging.json`.

```
{
  "compression": {
    "algorithm": "zip",
    "level": "best",
    "fileExtension": ".zip"
  }
}
```

Figure 77

Compression

Compression reduces the size of the package, allowing for more efficient storage and transmission. The Zip algorithm, as defined by the ISO document compression standard, is used.

Requirements and Specifications

Algorithm	Zip
ISO Standard Compliance	The compression must adhere to the ISO/IEC 21320-1:2015 standard for document compression.
Compression Level	Configurable levels of compression (e.g., no compression, fastest, best compression).
File Extensions	Compressed files should use the `.zip` extension.

Configuration Details in `packaging.json`

```
{
  "compression": {
    "algorithm": "Zip",
    "standard": "ISO/IEC 21320-1:2015",
    "level": "best",
    "fileExtension": ".zip"
  }
}
```

Figure 78

Optimization

Optimization ensures that the Capsium package is delivered efficiently and performs optimally in various environments. This section describes the methods and attributes involved in optimizing content delivery for Capsium packages.

Content Delivery Optimization

Content Delivery Optimization focuses on improving the speed and efficiency with which package content is delivered to end-users. This includes techniques for minimizing load times, reducing bandwidth usage, and enhancing overall user experience.

Minification	The process of removing unnecessary characters from code (such as whitespace, comments, and redundant formatting) to reduce file size without affecting functionality. This is commonly applied to HTML, CSS, and JavaScript files.
Compression	The use of algorithms to reduce the size of files for transmission over the network. Common methods include gzip and Brotli compression.
Caching	Storing copies of files in strategic locations (such as on a user's device or at various points in a content delivery network) to reduce load times and server requests.

Image Optimization	Techniques for reducing the file size of images without significantly compromising quality. This can include methods like resizing, format conversion, and compression.
Lazy Loading	A strategy for loading images and other resources only when they are needed, rather than all at once. This can significantly improve initial load times and overall performance.

Example:

```
{
    "optimization": {
        "minification": {
            "html": true,
            "css": true,
            "js": true
        },
        "compression": {
            "enabled": true,
            "method": "gzip"
        },
        "caching": {
            "enabled": true,
            "strategy": "aggressive"
        },
        "imageOptimization": {
            "enabled": true,
            "methods": ["resize", "compress"]
        },
        "lazyLoading": {
            "enabled": true,
            "elements": ["images", "videos"]
        }
    }
}
```

Figure 79

In this example, various optimization techniques are enabled and configured:

Minification	HTML, CSS, and JavaScript files will be minified.
Compression	Gzip compression is enabled for reducing file sizes during transmission.
Caching	An aggressive caching strategy is employed to store and serve content efficiently.
Image Optimization	Images will be resized and compressed to reduce their file sizes.
Lazy Loading	Images and videos will be loaded only when they come into view, reducing initial load times.

Attributes Summary

optimization	The root object for optimization configuration.	
	minification	Object containing minification settings.

- html (`boolean`): Enable or disable HTML minification.
- css (`boolean`): Enable or disable CSS minification.
- js (`boolean`): Enable or disable JavaScript minification.

compression: Object containing compression settings.

- enabled (`boolean`): Enable or disable compression.
- method (`string`): Specifies the compression method (e.g., `gzip`, `brotli`).

caching: Object containing caching settings.

- enabled (`boolean`): Enable or disable caching.
- strategy (`string`): Specifies the caching strategy (e.g., `aggressive`, `conservative`).

imageOptimization: Object containing image optimization settings.

- enabled (`boolean`): Enable or disable image optimization.
- methods (`array of string`): Specifies the methods used for image optimization (e.g., `resize`, `compress`).

lazyLoading: Object containing lazy loading settings.

- enabled (`boolean`): Enable or disable lazy loading.
- elements (`array of string`): Specifies the elements to apply lazy loading to (e.g., `images`, `videos`).

By implementing these optimization techniques, Capsium packages can deliver content more efficiently, providing a faster and smoother user experience. This structured approach ensures that content is not only high-quality and correct but also optimized for performance and delivery.

Capsium modules

General

The Capsium framework is partitioned into a small core and a set of optional modules. The core — defined by the clauses on the Capsium package, the Capsium reactor, and the management of Capsium packages — specifies the minimum that every conformant package, packager, and reactor shall implement: package layout, core metadata fields, the manifest, core routing and

auto-generation, datasets, self-containment, SHA-256 integrity, zip/`.cap` packaging, validation, route serving, and the Monitoring HTTP API.

Every additional capability is defined as a named module in a clause of its own. Modules are opt-in: an implementation claims exactly the modules it supports, à la carte, and each module carries its own additional requirements and its own conformance class ([compliance]), so that conformance to each capability can be tested independently.

Module identification

Each module is identified by a lowercase kebab-case string, its module identifier. The module identifier names the module's clause (`Module: <Name>`), its anchor, and its conformance class. This document defines the following modules:

Table 4 — Modules defined by this document

Identifier	Name	Summary	Conformance class	Clause
`signatures`	Digital signatures	RSA-SHA256 signing and verification of packages (X.509 or OpenPGP)	Capsium signatures	[module-signatures]
`encryption`	Encryption	AES-256-GCM package encryption and DEK/OpenPGP encrypted content	Capsium encryption	[module-encryption]
`layered-storage`	Layered storage	Overlay layers with a unified view, tombstones, and layer visibility	Capsium layered-storage	[module-layered-storage]
`composite`	Composite packages	Dependencies, store resolution, and route inheritance across packages	Capsium composite	[module-composite]
`authentication`	User authentication	Apache `passwd` and OAuth authentication, route access control	Capsium authentication	[module-authentication]
`handler-routes`	Handler routes	HTTP API routes that bind a method and path to an executable handler	Capsium handler-routes	[module-handler-routes]
`testing`	Testing	The YAML test DSL for routes, files, data, and configuration	Capsium testing	[module-testing]
`registries`	Registries	Static registries for publishing, resolving, and installing packages	Capsium registries	[module-registries]
`writable-packages`	Writable packages	Reactor-side overlay writes, dataset CRUD and GraphQL APIs, and save	Capsium writable-packages	[module-writable-packages]

Claiming modules

A package, packager, or reactor that claims a module shall satisfy all of the numbered additional requirements of that module's clause, in addition to the requirements of the core. Claiming a module is all-or-nothing: partial support of a module's requirements does not constitute a claim.

A Capsium package may declare the modules it claims in `metadata.json` with the optional `modules` key (Clause 5, Metadata file), whose value is an array of module identifiers:

```
{
    "name": "example-capsium-package",
    "version": "1.0.0",
    "modules": ["signatures", "layered-storage"]
```

```
}
```

Figure 80

Module: Digital signatures

The signatures module gives a Capsium package authenticity and tamper evidence on top of the integrity checks of the core package. A signed package carries a digital signature that a recipient can verify with the publisher's public key, confirming both the identity of the signer and that the package has not been altered since it was signed.

Signing is optional: a conformant core package, packager, or reactor works without it. A package, packager, or reactor that claims the `signatures` module shall satisfy the additional requirements of this clause.

Additional requirements

1) A signed package shall carry a digital signature in a separate signature file (e.g., `signature.sig`) stored inside the package.
2) The signature shall be calculated over the canonical payload: the keys of `security.integrityChecks.checksums` taken in sorted order, with the bytes of each referenced file concatenated in that order.
3) The signature algorithm shall be RSA with SHA-256, with a minimum key length of 2048 bits.
4) The signing key shall be represented by an X.509 certificate or an OpenPGP key, as recorded in the `certificateType` of the `digitalSignatures` configuration of `security.json` (`"X.509"` or `"OpenPGP"`); verifiers shall auto-detect the verification procedure from `certificateType`. When `certificateType` is absent, the signature shall be treated as an X.509 signature.
5) The `digitalSignatures` configuration of `security.json` shall record the `publicKey` and `signatureFile` paths, and the public key or certificate shall be embedded in the package.
6) For OpenPGP signatures, the signature file shall contain an armored detached RSA-SHA256 OpenPGP signature over the same canonical payload, and the signer's public key shall be embedded in the package (e.g., `signature.pub.asc`).
7) A verifying party shall use the public key to verify the digital signature, and the verification process should ensure that the package has not been altered since it was signed; a package whose signature does not verify should be rejected.

Signature configuration in `security.json`

Requirements and specifications

1) Digital signature configuration:
   - `publicKey` — Path to the public key file (X.509 certificate or OpenPGP public key) used for verifying the digital signature, relative to the package root.
     - Type: `string`
     - Example: `"signature.pub.asc"`
   - `signatureFile` — Path to the signature file that contains the digital signature.
     - Type: `string`
     - Example: `"signature.sig"`
   - `certificateType` — The kind of signing key.
     - Type: `string`
     - Enumeration: `"X.509"`, `"OpenPGP"` (absent is equivalent to `"X.509"`)
2) Verification process:
   - The verifier shall select the verification procedure from `certificateType` (auto-detection).
   - The system must use the embedded public key to verify the digital signature over the canonical payload.

— The verification process should ensure that the package has not been altered since it was signed.

Use case	Package distribution	When distributing the Capsium package, the publisher signs the package with their private key. The recipient can then verify the package using the included public key to ensure it has not been tampered with.

Example configuration (X.509)

```
{
    "security": {
        "digitalSignatures": {
            "certificateType": "X.509",
            "publicKey": "keys/public.pem",
            "signatureFile": "signature.sig"
        }
    }
}
```

Figure 81

Example configuration (OpenPGP)

```
{
    "security": {
        "digitalSignatures": {
            "certificateType": "OpenPGP",
            "publicKey": "signature.pub.asc",
            "signatureFile": "signature.sig"
        }
    }
}
```

Figure 82

Signed payload and signature files

Digital signatures ensure the authenticity and integrity of the package, verifying that it has not been tampered with and confirming the identity of the signer. Capsium packages can use either X.509 certificates or OpenPGP keys for digital signatures.

The signed payload is constructed identically by every implementation: take the keys of `security.integrityChecks.checksums` in sorted order and concatenate the bytes of each referenced file in that order. The signature is calculated over that byte stream with RSA-SHA256 (openssl interop: `openssl dgst -sha256 -sign` / `-verify`).

The checksums are computed before signing: they cover every package file except `security.json` itself and the signature file, which is added to the package afterwards. The embedded public key is ordinary package content and is covered by the checksums.

Signature file formats:

X.509 — `signature.sig` holds the raw RSA-SHA256 signature bytes; the public key or certificate travels inside the package at the `publicKey` path.

OpenPGP `signature.sig` holds an armored detached RSA-SHA256 OpenPGP signature over the same canonical payload; the signer's armored public key is embedded in the package (e.g., `signature.pub.asc`).

When encryption is also applied ([module-encryption]), encryption happens before signing.

Requirements and specifications

Signature algorithm	RSA with SHA-256	
Key length	Minimum 2048 bits	
Digital certificate	X.509	Must follow the X.509 standard for public key infrastructure.
	OpenPGP	Must follow the OpenPGP standard for encryption and signatures.
Signature file	A separate file (e.g., `signature.sig`) containing the digital signature.	

Conformance

A Capsium package / packager / reactor conforming to class `signatures` SHALL satisfy requirements signatures-1 to signatures-7 of this clause. The conformance class is defined in [conformance-signatures].

Module: Encryption

The encryption module protects the content of a Capsium package from unauthorized access, ensuring that only intended recipients can decrypt and access the data. A Capsium package can contain both encrypted and cleartext content; encryption uses public/private keys and data encryption keys (DEK), or OpenPGP, and the whole package payload can be encrypted with AES-256 in GCM mode.

Encryption is optional: a conformant core package, packager, or reactor works without it. A package, packager, or reactor that claims the `encryption` module shall satisfy the additional requirements of this clause.

Additional requirements

1) Package encryption shall use the AES-256 algorithm with GCM (Galois/Counter Mode) for authenticated encryption, and the encrypted package file shall use the `.enc` extension.
2) Encryption happens before signing ([module-signatures]); the files `metadata.json` and `signature.json` itself are unencrypted.
3) Encryption keys shall be distributed and stored securely (key management).
4) When files are encrypted individually, the package shall carry an `encryption` configuration with a `publicKeyFile` and an `encryptedFiles` list whose entries declare `file`, `encryptedWith` (`"DEK"` or `"OpenPGP"`), and `algorithm` (`"OCB"` or `"OpenPGP"`, matching the encryption method).
5) A data encryption key (DEK) shall be encrypted with the recipient's public key — wrapped with RSA-OAEP-SHA256 or sent as an armored OpenPGP message — so that only the holder of the corresponding private key can decrypt the DEK and subsequently the files.
6) The encryption envelope shall be recorded in `signature.json` with `algorithm` `"AES-256-GCM"`, a `keyManagement` of `"RSA-OAEP-SHA256"` or `"OpenPGP"`, the base64-encoded `iv` and `authTag` of the GCM ciphertext, and the protected DEK: either `encryptedDek` (base64 RSA-OAEP-SHA256-wrapped DEK) or `message` (armored OpenPGP message containing the DEK). Decryptors shall auto-detect the key management procedure from `keyManagement`.

7) Routes that serve encrypted files should indicate that the files are encrypted and specify the required decryption method, and the manifest should list encrypted files with their encryption details.

Package encryption

Encryption protects the package's contents from unauthorized access, ensuring that only intended recipients can decrypt and access the data.

Encryption happens before signing. The file `metadata.json` and `signature.json` itself are unencrypted.

When an encrypted and signed Capsium package is uncompressed, it looks like this:

— package/metadata.json
— package/signature.json
— package/package.enc

When `package.enc` is decrypted, the package looks like this, depending on what the package contains:

— package/metadata.json
— package/signature.json
— package/routes.json
— package/manifest.json
— package/storage.json
— package/contents/index.html
— package/data/my_yaml.yaml

Requirements and Specifications

Encryption Algorithm	AES-256
Mode of Operation	GCM (Galois/Counter Mode) for authenticated encryption
Key Management	RSA-OAEP-SHA256 wrapping of the data encryption key (DEK), or an armored OpenPGP message containing the DEK
Encrypted File	The encrypted package should have a `.enc` extension.

Encryption envelope in `signature.json`

The `signature.json` file of an encrypted package carries the cleartext encryption envelope: the algorithm, the key management procedure, the GCM parameters, and the protected DEK. The DEK is 32 random bytes; the `iv` is the base64-encoded 12-byte GCM initialization vector and `authTag` the base64-encoded 16-byte GCM authentication tag.

RSA-OAEP-SHA256 key management — the DEK is wrapped with the recipient's RSA public key (OAEP with SHA-256 and MGF1-SHA256):

```
{
  "encryption": {
    "algorithm": "AES-256-GCM",
    "keyManagement": "RSA-OAEP-SHA256",
    "encryptedDek": "<base64 RSA-OAEP-SHA256-wrapped data-encryption key>",
    "iv": "<base64>",
    "authTag": "<base64>"
  }
```

```
}
```

Figure 83

OpenPGP key management — the DEK is sent as an armored OpenPGP message to the recipient's OpenPGP public key:

```
{
  "encryption": {
    "algorithm": "AES-256-GCM",
    "keyManagement": "OpenPGP",
    "message": "<armored OpenPGP message containing the data-encryption key>",
    "iv": "<base64>",
    "authTag": "<base64>"
  }
}
```

Figure 84

Decryptors auto-detect the key management procedure from `keyManagement`: given the corresponding private key, they recover the DEK from `encryptedDek` (RSA-OAEP-SHA256) or from `message` (OpenPGP) and decrypt `package.enc` transparently. `metadata.json` stays cleartext for identification.

Encrypted information

General

The Capsium package can contain both encrypted and cleartext content. Encryption uses public/private keys and data encryption keys (DEK) with the OCB algorithm or OpenPGP to ensure security.

This section details the requirements and specifications for each aspect of encryption, along with relevant use cases and how encrypted files interact with `routes.json` and `manifest.json`.

Example:

```
{
    "encryption": {
        "publicKeyFile": "path/to/public.key",
        "encryptedFiles": [
            {
                "file": "secret.dat",
                "encryptedWith": "DEK",
                "algorithm": "OCB"
            }
        ]
    }
}
```

Figure 85

Public Key File

Requirements and Specifications	1)	publicKeyFile:	
		Description	Path to the public key file used for encrypting the Data Encryption Key (DEK) or directly encrypting files using OpenPGP.
		Type	`string`

	Value Requirements	— Must be a valid file path. — The file should exist and be accessible. — Example: `"path/to/public.key"`
Use Case	Encrypting DEK	When encrypting sensitive files, the DEK is encrypted using the recipient's public key to ensure that only the recipient, who possesses the corresponding private key, can decrypt the DEK and subsequently the files.
	OpenPGP Encryption	Files can be directly encrypted using OpenPGP with the recipient's public key, ensuring secure transmission and storage.

Example Configuration

```
{
    "encryption": {
        "publicKeyFile": "path/to/public.key"
    }
}
```

Figure 86

Encrypted Files

Requirements and Specifications	1)	encryptedFiles:				
		Description	List of files within the package that are encrypted.			
		Type	`array`			
		Items	file	Description:	Path to the encrypted file within the package.	
				Type:	`string`	
				Value Requirements:	— Must be a valid file path. — The file should exist and be part of the package. — Example: `"secret.dat"`	
			encryptedWith	Description:	Indicates the method used for encryption.	
				Type:	`string`	
				Enumeration:	`"DEK"`, `"OpenPGP"`	
				Value Requirements:	— Must be	

"DEK" or "OpenPGP".
- "DEK" specifies the use of Data Encryption Key.
- "OpenPGP" specifies the use of OpenPGP encryption.

	algorithm	Description: The encryption algorithm used. Type: string Enumeration: "OCB", "OpenPGP" Value Requirements: - Must be "OCB" when encryptedWith is "DEK". - Must be "OpenPGP" when encryptedWith is "OpenPGP".

	Use Case	Sensitive File Encryption with DEK	To protect sensitive data within a package, files like `secret.dat` are encrypted using a DEK. The DEK is then encrypted with the recipient's public key to ensure secure transmission.
		Sensitive File Encryption with OpenPGP	Files can be directly encrypted using OpenPGP with the recipient's public key, providing an alternative method for secure file encryption.

Example Configuration

```
{
    "encryption": {
        "encryptedFiles": [
            {
                "file": "secret.dat",
                "encryptedWith": "DEK",
                "algorithm": "OCB"
            },
            {
```



```
                "file": "confidential.txt",
                "encryptedWith": "OpenPGP",
                "algorithm": "OpenPGP"
            }
        ]
    }
}
```

Figure 87

Interaction with `routes.json` and `manifest.json`

Requirements and Specifications

1) routes.json:

Description	This file defines the routing of various endpoints within the package.
Handling Encrypted Files	Routes that serve encrypted files should indicate that the files are encrypted and specify the decryption method required.
Example Configuration	

```
{
    "routes": [
        {
            "path": "/download/secret",
            "file": "secret.dat",
            "encrypted": true,
            "decryptionMethod": "DEK"
        },
        {
            "path": "/download/confidential",
            "file": "confidential.txt",
            "encrypted": true,
            "decryptionMethod": "OpenPGP"
        }
    ]
}
```

Figure 88

1) manifest.json:

Description	This file contains metadata about the package, including information about the encrypted files.
Handling Encrypted Files	The manifest should list encrypted files and provide details about their encryption methods.
Example Configuration	

```
{
    "manifestVersion": "1.0",
    "description": "Capsium package containing both encrypted and cleartext
content.",
    "files": [
        {
            "path": "secret.dat",
            "encrypted": true,
            "encryptionDetails": {
                "encryptedWith": "DEK",
                "algorithm": "OCB"
            }
```

```
        },
        {
            "path": "confidential.txt",
            "encrypted": true,
            "encryptionDetails": {
                "encryptedWith": "OpenPGP",
                "algorithm": "OpenPGP"
            }
        },
        {
            "path": "readme.txt",
            "encrypted": false
        }
    ]
}
```

Figure 89

Detailed Use Cases and Examples

Use Case: Serving Encrypted Files via `routes.json`

When a client requests a file that is listed in `routes.json`, the server identifies if the file is encrypted based on the `encrypted` attribute. It then uses the specified `decryptionMethod` to decrypt the file before serving it to the client.

Example Entry in `routes.json`

```
{
    "routes": [
        {
            "path": "/download/secret",
            "file": "secret.dat",
            "encrypted": true,
            "decryptionMethod": "DEK"
        },
        {
            "path": "/download/confidential",
            "file": "confidential.txt",
            "encrypted": true,
            "decryptionMethod": "OpenPGP"
        }
    ]
}
```

Figure 90

Explanation
- The route `/download/secret` serves the `secret.dat` file, which is encrypted using a DEK and needs to be decrypted using the DEK method.
- The route `/download/confidential` serves the `confidential.txt` file, which is encrypted using OpenPGP and must be decrypted using the OpenPGP method.

Use Case: Metadata Management in `manifest.json`

The `manifest.json` provides a comprehensive overview of the files in the package, indicating which files are encrypted and detailing the encryption methods used. This helps clients understand how to handle and decrypt the files correctly.

Example Entry in `manifest.json`

```
{
    "manifestVersion": "1.0",
    "description": "Capsium package containing both encrypted and cleartext
content.",
    "files": [
        {
            "path": "secret.dat",
            "encrypted": true,
            "encryptionDetails": {
                "encryptedWith": "DEK",
                "algorithm": "OCB"
            }
        },
        {
            "path": "confidential.txt",
            "encrypted": true,
            "encryptionDetails": {
                "encryptedWith": "OpenPGP",
                "algorithm": "OpenPGP"
            }
        },
        {
            "path": "readme.txt",
            "encrypted": false
        }
    ]
}
```

Figure 91

Explanation — The `secret.dat` file is marked as encrypted with details specifying it uses a DEK and the OCB algorithm.
— The `confidential.txt` file is marked as encrypted with details specifying it uses OpenPGP.
— The `readme.txt` file is not encrypted.

Value Requirements and Enumerations for Attributes

1) publicKeyFile:
   Type `string`
   Value Requirements Valid file path, accessible.
2) encryptedFiles:
   Type `array`
   Items
   - file — Type `string`; Value Requirements: Valid file path, part of the package.
   - encryptedWith — Type `string`; Enumeration: `"DEK"`, `"OpenPGP"`; Value Requirements: Must be either `"DEK"` or `"OpenPGP"`.
   - algorithm — Type `string`; Enumeration: `"OCB"`, `"OpenPGP"`; Value Requirements: Must match the encryption method.

3) routes.json:
   Attributes
   — `path`: `string` (Valid route path)
   — `file`: `string` (Valid file path)
   — `encrypted`: `boolean`
   — `decryptionMethod`: `string` (Enumeration: `"DEK"`, `"OpenPGP"`)
4) manifest.json:
   Attributes
   — `manifestVersion`: `string`
   — `description`: `string`
   — `files`: `array`
     Items
     — `path`: `string` (Valid file path)
     — `encrypted`: `boolean`
     — `encryptionDetails`: `object` (Required if `encrypted` is `true`)

Attributes in `manifest.json`

1) manifestVersion:
   Type `string`
   Description Version of the manifest schema.
   Example `"1.0"`
2) description:
   Type `string`
   Description A description of the Capsium package.
   Example `"Capsium package containing both encrypted and cleartext content."`
3) files:
   Type `array`
   Description List of files included in the package.
   Items
   path — Type: `string`; Description: Path to the file within the package.; Example: `"secret.dat"`
   encrypted — Type: `boolean`; Description: Indicates whether the file is encrypted.; Example: `true` if the file is encrypted, `false` otherwise.
   encryptionDetails — Type: `object`; Description: Details about the encryption method used. Required if `encrypted` is `true`.
   Properties: `encryptionMethod` (Type: `string`; Enumeration: `"DEK"`, `"OpenPGP"`; Description: The method used for encryption.; Example: `"DEK"` or `"OpenPGP"`), `algorithm` (Type: `string`; Enumeration: `"GCM"`, `"OpenPGP"`; Description: The algorithm used for encryption.; Example: `"GCM"`), `keyDerivation` (Type: `object`; Description: Key derivation details.; Properties: `algorithm` (Type: `string`; Enumeration: `"PBKDF2"`, `"OpenPGP"`; Description: The key derivation algorithm used.; Example: `"PBKDF2"`), `iterations` (Type: `integer`; Description: The number of iterations used in key derivation.; Example: `100000`), `salt` (Type: `string`; Description: The salt used in key derivation, encoded in base64.; Example: `"c2FsdHNhbHRzYWx0"`)), `iv` (Type: `string`; Description: The initialization vector used for encryption, encoded in base64.; Example: `"aXZpdml2aXZpdg=="`), `tag` (Type: `string`; Description: The authentication tag used for verifying the integrity of the encrypted data, encoded in base64.; Example: `"dGFndGFndGFndGFn"`), `publicKeyId` (Type: `string`; Description: The ID of the public key used for encryption, if applicable.; Example: `"0x1234567890ABCDEF"`)

The encryption algorithm used. "OCB" or "OpenPGP"

Description: Example:

Example Configuration in `manifest.json`

```
{
    "manifestVersion": "1.0",
    "description": "Capsium package containing both encrypted and cleartext
content.",
    "files": [
        {
            "path": "secret.dat",
            "encrypted": true,
            "encryptionDetails": {
                "encryptedWith": "DEK",
                "algorithm": "OCB"
            }
        },
        {
            "path": "confidential.txt",
            "encrypted": true,
            "encryptionDetails": {
                "encryptedWith": "OpenPGP",
                "algorithm": "OpenPGP"
            }
        },
        {
            "path": "readme.txt",
            "encrypted": false
        }
    ]
}
```

Figure 92

Detailed Example

Combined Example Configuration

Here is a comprehensive example showing how the `encryption`, `routes.json`, and `manifest.json` files work together in a Capsium package.

Encryption Configuration (`encryption` section)

```
{
    "encryption": {
        "publicKeyFile": "path/to/public.key",
        "encryptedFiles": [
            {
                "file": "secret.dat",
                "encryptedWith": "DEK",
                "algorithm": "OCB"
            },
```

```
            {
                "file": "confidential.txt",
                "encryptedWith": "OpenPGP",
                "algorithm": "OpenPGP"
            }
        ]
    }
}
```

Figure 93

Routes Configuration (`routes.json`)

```
{
    "routes": [
        {
            "path": "/download/secret",
            "file": "secret.dat",
            "encrypted": true,
            "decryptionMethod": "DEK"
        },
        {
            "path": "/download/confidential",
            "file": "confidential.txt",
            "encrypted": true,
            "decryptionMethod": "OpenPGP"
        },
        {
            "path": "/download/readme",
            "file": "readme.txt",
            "encrypted": false
        }
    ]
}
```

Figure 94

Manifest Configuration (`manifest.json`)

```
{
    "manifestVersion": "1.0",
    "description": "Capsium package containing both encrypted and cleartext
content.",
    "files": [
        {
            "path": "secret.dat",
            "encrypted": true,
            "encryptionDetails": {
                "encryptedWith": "DEK",
                "algorithm": "OCB"
            }
        },
        {
            "path": "confidential.txt",
            "encrypted": true,
            "encryptionDetails": {
                "encryptedWith": "OpenPGP",
                "algorithm": "OpenPGP"
            }
        },
        {
            "path": "readme.txt",
            "encrypted": false
```



```
        }
    ]
}
```

Figure 95

Conformance

A Capsium package / packager / reactor conforming to class `encryption` SHALL satisfy requirements encryption-1 to encryption-7 of this clause. The conformance class is defined in [conformance-encryption].

Module: Layered storage

The layered storage module combines multiple storage layers into a single, unified filesystem view, similar to the functionality of "overlay FS". This approach ensures that changes can be made in a non-destructive manner while preserving the original data of a package.

Layered storage is optional: the core package serves its content and datasets without it. A package, packager, or reactor that claims the `layered-storage` module shall satisfy the additional requirements of this clause.

Additional requirements

1) The storage system shall support multiple layers stacked in a declared order; the layers of a package shall be declared in `storage.json` under `storage.layers` as an array ordered from bottom to top, each layer declaring its `path`, whether it is `writable`, and its `visibility`.
2) The `content/` directory of the package shall always constitute the implicit bottom layer, even when `storage.layers` is declared; every declared layer is a package-relative directory that mirrors the `content/` tree.
3) The system shall present a single filesystem view that merges all layers; resolution shall proceed from the top layer to the bottom layer, and the first hit wins.
4) A content path that misses all of the package's own layers shall fall through to the exported content of its dependencies ([module-composite]): content of the package itself shadows dependency content at the same path.
5) Deletions shall be recorded as tombstones in a `.capsium-tombstones` file — a JSON array of content-relative paths — kept in a writable layer; tombstones shall be honored at and below their layer, so that the tombstones of a dependent package also suppress dependency content beneath them.
6) Runtime marker files — the `.capsium-tombstones` file itself — shall not be covered by pack-time checksums (Clause 5, Security); other dotfiles (for example `.htpasswd`) are normal package content and shall be covered when packed.
7) Resolution through layers shall apply to content paths only: non-content paths, such as dataset sources under `data/`, shall bypass the layers and resolve at the package root.
8) The `visibility` field shall determine whether the layer is exposed as an inheritable interface (`exported`) or kept private (`private`); a layer marked `private` shall be invisible to dependent packages ([module-composite]).

Layer structure and configuration

Each layer is a package-relative directory that mirrors the `content/` tree: a file at `<layer>/index.html` shadows or augments `content/index.html`. Layers are declared in `storage.json` under `storage.layers`, ordered from bottom to top, and the `content/` directory always sits beneath them as the implicit bottom layer:

```
{
  "storage": {
    "layers": [
      {
```

```
        "path": "base",
        "writable": false,
        "visibility": "exported"
      },
      {
        "path": "updates",
        "writable": true,
        "visibility": "private"
      }
    ]
  }
}
```

Figure 96

In this example the effective stack, from bottom to top, is `content/` (implicit bottom layer), then `base/`, then `updates/`. A package without a `layers` configuration behaves as a single implicit layer — its `content/` directory.

layer attributes	`path`	Package-relative directory of the layer, mirroring the `content/` tree.
	`writable`	Whether a reactor may write changes into the layer. Writes — performed by reactors claiming the writable packages module ([module-writable-packages]) — go only to the topmost writable layer, preserving lower layers intact.
	`visibility`	`exported` layers are visible to dependent packages as an inheritable interface; `private` layers are invisible to dependent packages ([module-composite]).

Merged resolution

The merged view resolves a content path against the stack from the top layer to the bottom layer; the first layer containing the path wins. A content path that is present in none of the package's own layers falls through to the exported content of the package's dependencies ([module-composite]), so a package transparently serves dependency content it does not shadow. Resolution through layers applies to content paths only: non-content paths, such as dataset sources under `data/`, bypass the layers and resolve directly at the package root.

Tombstones

Deletions do not physically remove files from lower layers. Instead, the deleting layer records a tombstone in a `.capsium-tombstones` file: a JSON array of content-relative paths, kept at the root of the writable layer:

```
["index.html", "images/legacy-banner.png"]
```

Figure 97

A tombstoned path resolves as absent — requests to it are answered `404 Not Found` — even when a lower layer still contains the file. Tombstones are honored at and below their own layer: the tombstones recorded by a dependent package therefore also suppress dependency content that would otherwise fall through to it.

The `.capsium-tombstones` file is a runtime marker file: it is written and rewritten by reactors at runtime and is not covered by the pack-time checksums of `security.json` (Clause 5, Security). This exception is narrow: any other dotfile — for example an `.htpasswd` file

([module-authentication]) — is normal package content, packed and covered by checksums like any other file.

Conformance

A Capsium package / packager / reactor conforming to class `layered-storage` SHALL satisfy requirements layered-storage-1 to layered-storage-8 of this clause. The conformance class is defined in [conformance-layered-storage].

Module: Composite packages

The composite packages module combines multiple Capsium packages into a single, cohesive unit: a package declares dependencies on other packages, resolves them from a bundle, a store, or a registry, and re-uses their exported content and routes. Dependencies may also be encapsulated — embedded as `.cap` files inside the dependent package — so that the whole unit is distributed as one artifact.

The field definitions this module builds on remain in the core clause: dependencies are declared with the `dependencies` key of `metadata.json` (Clause 5.3.4), immutability with the `readOnly` key (Clause 5.3.6), resource visibility in `manifest.json` (Clause 5.4.2), and route visibility in `routes.json` (Clause 5.7.6). A package, packager, or reactor that claims the `composite` module shall satisfy the additional requirements of this clause.

Additional requirements

1) Dependencies between packages shall be declared in `metadata.json` as an object mapping the GUID of each required package to a semantic version range (Clause 5.3.4).
2) A dependency resource reference shall take the form `<guid>/<path>`, where `<guid>` is any absolute URI (`capsium:`, `https:`, …) and `<path>` is a package-relative path within the referenced package; when several declared dependency GUIDs are a prefix of the reference, the longest GUID prefix shall win.
3) Dependency packages shall be resolved through the chain *bundle → store → registry*: first among the packages encapsulated in the dependent package, then in the package store, then in a configured registry ([module-registries]). For each dependency, the newest available version satisfying the declared range shall be selected.
4) A package store shall be a directory containing `<name>-<version>.cap` files, optionally with an `index.json` mapping package GUIDs to store files.
5) A package may encapsulate its dependencies by embedding their `.cap` files under `packages/` and recording them in `packages/index.json`, mapping each GUID to its `file`, `version`, and `sha256`. Encapsulated files shall be covered by the parent package's checksums; the recorded `sha256` shall be re-verified at resolution; each encapsulated package's own `security.json` shall be verified at activation. The parent package should declare the transitive closure of the dependencies it bundles (one-level policy).
6) On activation of a composite package, the `content/` of each resolved dependency shall become a lower read-only layer below all of the dependent's own layers ([module-layered-storage]), and only `exported` resources and routes shall be visible to the dependent; a reference to a `private` resource or route of a dependency is an error.
7) Inherited routes shall be declared in the `routes.json` of the dependent package, referencing the dependency resource (composite-2); the dependent may remap the route, rewrite its response, enhance its response headers, and supplant its request headers, with the semantics of this clause.
8) Each package within the composite package should have its own digital signature, and the composite package itself should also be signed ([module-signatures]); SHA-256 checksums shall be used to verify that the package contents have not been tampered with.

Structure

A composite package is an ordinary Capsium package that declares dependencies and, optionally, encapsulates the resolved dependency `.cap` files under `packages/` with an index:

```
composite-package/
├── metadata.json          # dependencies: guid -> semver range
├── manifest.json
├── routes.json            # may reference dependency resources
├── storage.json
├── content/
└── packages/              # encapsulated dependencies (optional)
    ├── index.json
    ├── core-lib-1.0.0.cap
    └── theme-pack-2.1.0.cap
```

Figure 98

`packages/index.json` maps each encapsulated GUID to its file, version, and SHA-256 checksum:

```
{
  "packages": {
    "capsium://example.com/core-lib": {
      "file": "packages/core-lib-1.0.0.cap",
      "version": "1.0.0",
      "sha256": "9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08
"
    },
    "https://packages.example.org/theme-pack": {
      "file": "packages/theme-pack-2.1.0.cap",
      "version": "2.1.0",
      "sha256": "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
"
    }
  }
}
```

Figure 99

Encapsulated `.cap` files are package content of the parent: they are covered by the parent package's pack-time checksums (Clause 5, Security), and the `sha256` recorded in `packages/index.json` is re-verified whenever the encapsulated package is resolved. At activation, each encapsulated package's own `security.json` is verified like that of any other package.

A parent package should declare the transitive closure of the dependencies it bundles: every dependency of an encapsulated package should itself be declared — and resolved — by the parent (one-level policy).

Dependency resolution

Dependency resolution selects, for every GUID declared in `metadata.dependencies`, the newest available version satisfying the declared range, and makes it available to the dependent package. Sources are consulted in order:

1) Bundle:: the packages encapsulated under `packages/` (composite-5);
2) Store:: a package store directory — for example from the `CAPSIUM_STORE` environment variable or a `--store` option — containing `<name>-<version>.cap` files and an optional `index.json` mapping GUIDs to store files;

3) Registry:: a static registry ([module-registries]), from which the selected version is installed into the store.

Resolution shall select the newest version satisfying the declared semantic version range. Management tooling should detect unresolvable dependencies — a missing GUID or a range no available artifact satisfies — and report them before activation ([management-clause], Dependency resolution).

Dependency resource references

A route, resource, or mount of a dependent package references a resource of a dependency with a reference of the form `<guid>/<path>`:

— `<guid>` is the absolute-URI GUID of the dependency as declared in `metadata.dependencies` — any absolute URI scheme may be used (`capsium://example.com/core-lib`, `https://packages.example.org/theme-pack`, …);
— `<path>` is the package-relative path of the resource within the referenced package, for example `content/app.js`.

When several declared GUIDs are a prefix of one another, the reference shall be matched against the longest GUID prefix.

Example:

```
{
  "path": "/vendor/core/app.js",
  "resource": "capsium://example.com/core-lib/content/app.js"
}
```

Figure 100

Only `exported` resources and routes of a dependency may be referenced this way (Clause 5.4.2, Clause 5.7.6); referencing a `private` resource or route of a dependency is an error.

Layer inheritance

On activation of a composite package, the `content/` directory of each resolved dependency becomes a lower read-only layer, below all of the dependent package's own layers, and merged resolution follows the rules of the layered storage module ([module-layered-storage]): the dependent's own content shadows dependency content, dependency content serves paths the dependent does not provide, and tombstones recorded by the dependent suppress dependency content beneath them. Layers a dependency marks `private` are not exposed to the dependent.

Route inheritance and processing

The dependent package inherits routes from its dependencies by declaring them in its own `routes.json` with a dependency resource reference (composite-2). Only routes marked `exported` can be inherited (Clause 5.7.6). Inherited routes may be processed with the following route attributes, whose semantics are:

`remap`	The effective path at which the route is served, replacing the declared `path` for serving.
`responseRewrite`	A replacement of the response: the declared `body` replaces the response content of the inherited route wholesale, and the declared `headers` are merged per-key into the response headers.
`responseHeaders`	Headers merged over the serving defaults: each declared header is set on the response, overriding the reactor's default for that header

— including the default `Cache-Control` ([reactor-clause]) — rather than being applied only when absent.

`requestHeaders` — Headers applied to the request when it is forwarded to handler execution ([module-handler-routes]); for static resource serving they have no effect.

Example `routes.json` of a dependent package:

```
{
  "routes": [
    {
      "path": "/api/exported",
      "resource": "capsium://example.com/core-lib/content/api/public.json",
      "visibility": "exported"
    },
    {
      "path": "/api/remapped",
      "resource": "capsium://example.com/core-lib/content/api/hidden.json",
      "visibility": "private",
      "remap": "/api/newpath"
    },
    {
      "path": "/api/rewritten",
      "resource": "capsium://example.com/core-lib/content/api/raw.json",
      "visibility": "private",
      "responseRewrite": {
        "body": "Modified response content",
        "headers": {
          "X-Custom-Header": "CustomValue"
        }
      }
    },
    {
      "path": "/api/enhanced",
      "resource": "capsium://example.com/core-lib/content/api/cached.json",
      "visibility": "exported",
      "responseHeaders": {
        "Cache-Control": "no-cache",
        "X-Enhanced-Header": "EnhancedValue"
      }
    },
    {
      "path": "/api/supplanted",
      "handler": "content/handlers/api.js",
      "method": "GET",
      "visibility": "private",
      "requestHeaders": {
        "Authorization": "Bearer new-token",
        "X-Forwarded-For": "client-ip"
      }
    }
  ]
}
```

Figure 101

By clearly defining and adhering to these visibility, inheritance, and processing rules, the Capsium package ensures that resource routing is both flexible and secure, enabling effective re-use of routes while protecting private routes and allowing for extensive customization.

Security, digital signatures, and integrity checks

Ensuring the security and integrity of a composite package involves digital signatures and integrity checks at two levels: the composite package itself and every package it encapsulates.

Integrity checks	The parent package's `security.json` covers every file of the parent, including the encapsulated `.cap` files; the `sha256` recorded for each encapsulated package in `packages/index.json` is re-verified at resolution, and each encapsulated package's own `security.json` is verified at activation (Clause 5, Security).
Digital signatures	Each package within the composite package should have its own digital signature, and the composite package itself should also be signed ([module-signatures]).

User authentication

A composite package may apply user authentication to its own routes and to inherited routes. The package-side authentication configuration is defined by the authentication module ([module-authentication]).

Conformance

A Capsium package / packager / reactor conforming to class `composite` SHALL satisfy requirements composite-1 to composite-8 of this clause. The conformance class is defined in [conformance-composite].

Module: User authentication

The authentication module secures access to the resources and datasets of an activated Capsium package. It covers user authentication methods — Apache `passwd` (`.htpasswd`) basic authentication and external OAuth — and the per-route access control declarations that restrict dataset routes to authenticated users with the required roles.

Authentication is optional: a conformant core package and reactor serve routes without it. A package, packager, or reactor that claims the `authentication` module shall satisfy the additional requirements of this clause.

Additional requirements

1) Each user shall have a unique user ID and a strong password; password policies should enforce a minimum length of 8 characters with at least one uppercase letter, one lowercase letter, one digit, and one special character.
2) Account lockout should be implemented after a specified number of failed login attempts (e.g., 5 attempts), and sessions should be managed securely with timeout and automatic logout after a period of inactivity.
3) All authentication data should be encrypted during transmission and storage.
4) For Apache `passwd` authentication, the `.htpasswd` file shall be securely stored and accessible only to the web server, and passwords should be hashed using a secure algorithm (e.g., bcrypt).
5) For OAuth, the client secret shall be kept protected from users who can extract the package: it shall be stored in a secure environment variable or a secure server-side configuration file, not within the package itself.

6) Dataset routes may declare `accessControl` with `roles` and `authenticationRequired`; a reactor claiming this module shall enforce those declarations on the affected routes.

Authentication methods

General

General user authentication requirements apply to all methods and provide a foundation for secure access control.

Example:

```
{
    "authentication": {
        "basicAuth": {
            "enabled": true,
            "passwdFile": "path/to/.htpasswd"
        },
        "oauth": {
            "enabled": true,
            "provider": "Google",
            "clientId": "your-client-id",
            "clientSecret": "your-client-secret",
            "redirectUri": "your-redirect-uri"
        }
    }
}
```

Figure 102

Requirements and Specifications

User ID and Password	Each user must have a unique User ID and a strong password.
Password Policy	Enforce strong password policies, including: — Minimum length: 8 characters — At least one uppercase letter, one lowercase letter, one digit, and one special character
Account Lockout	Implement account lockout after a specified number of failed login attempts (e.g., 5 attempts).
Session Management	Secure session management with timeout and automatic logout after a period of inactivity.
Encryption	All authentication data should be encrypted during transmission and storage.

Apache Authentication (passwd)

Apache authentication using `passwd` involves basic HTTP authentication with user credentials stored in a `.htpasswd` file.

Requirements and Specifications

File Location	The `.htpasswd` file must be securely stored and accessible only to the web server.
Encryption	Passwords in the `.htpasswd` file should be hashed using a secure algorithm (e.g., bcrypt).
Configuration	Apache configuration to support basic authentication using the `.htpasswd` file.

Configuration Details

htpasswd File::
— Use the `htpasswd` utility to create and manage the file.
— Example entry: `username:$apr1$eWvS2f3d$Ee9uU7/r8C3W1J9QkE45H0`
— **Apache Configuration** (`.htaccess` or `httpd.conf`): ` AuthType Basic AuthName “Restricted Access” AuthUserFile /path/to/.htpasswd Require valid-user `

External Authentication via OAuth

External authentication via OAuth involves delegating the authentication process to an external OAuth provider (e.g., Google, Facebook).

Requirements and Specifications

OAuth Provider	Select a trusted OAuth provider (e.g., Google, Facebook, GitHub).
Client ID and Secret	Obtain a Client ID and Secret from the OAuth provider.
Redirect URI	Configure a redirect URI on the OAuth provider’s dashboard that points to your application’s OAuth callback endpoint.
Scope	Define the scope of access (e.g., email, profile).
State Parameter	Use the state parameter to prevent CSRF attacks.
Token Management	Securely manage and store OAuth tokens.
Secret Protection	The OAuth secret must be kept protected from users who can extract the package. This includes storing the secret in a secure environment variable or a secure server-side configuration file, not within the package itself.

Configuration Details in `oauth_config.json`

`json { “oauth”: { “provider”: “Google”, “clientId”: “YOUR_CLIENT_ID”, “clientSecret”: “${OAUTH_CLIENT_SECRET}”, “redirectUri”: “https://yourapp.com/oauth/callback”, “scope”: [“email”, “profile”], “stateSecret”: “YOUR_STATE_SECRET” } } `

Secret Protection

To protect the OAuth secret from users who can extract the package: - Store the `clientSecret` in a secure environment variable (`OAUTH_CLIENT_SECRET`) instead of hardcoding it in the configuration file. - Ensure that the `oauth_config.json` file references the environment variable for the

`clientSecret`. - On the server side, configure the environment variable securely and load it during application startup.

Combined Configuration Example

Here is a combined configuration example that includes general requirements, Apache authentication, and OAuth configuration.

`authentication.json`

`json { “authentication”: { “general”: { “passwordPolicy”: { “minLength”: 8, “requireUppercase”: true, “requireLowercase”: true, “requireDigit”: true, “requireSpecialCharacter”: true }, “accountLockout”: { “threshold”: 5, “duration”: 30 }, “sessionManagement”: { “timeout”: 30, “autoLogout”: true }, “encryption”: “AES-256” }, “apache”: { “htpasswdFile”: “/path/to/.htpasswd”, “encryptionAlgorithm”: “bcrypt” }, “oauth”: { “provider”: “Google”, “clientId”: “YOUR_CLIENT_ID”, “clientSecret”: “${OAUTH_CLIENT_SECRET}”, “redirectUri”: “https://yourapp.com/oauth/callback”, “scope”: [“email”, “profile”], “stateSecret”: “YOUR_STATE_SECRET” } } } `

general Specifies general authentication requirements, including password policy, account lockout, session management, and encryption.

apache Details Apache authentication configuration, including the path to the `.htpasswd` file and the encryption algorithm used for passwords.

oauth Configures external authentication via OAuth, including the OAuth provider, client ID, client secret (referenced from an environment variable), redirect URI, scope, and state secret.

This comprehensive configuration ensures that user authentication is secure, flexible, and compliant with best practices, while also protecting sensitive information such as the OAuth secret from being exposed.

Dataset route access control

Dataset routes (Clause 5.7.3) may carry access control configurations to manage who can access the data and under what conditions. Access control is declared per route in `routes.json` with the `accessControl` key.

Example of `routes.json`:

```
{
    "routes": [
        {
            "route": "/api/v1/data/users",
            "dataset": "users",
            "accessControl": {
                "roles": ["admin", "user"],
                "authenticationRequired": true
            }
        },
        {
            "route": "/api/v1/data/products",
            "dataset": "products",
            "accessControl": {
                "roles": ["admin"],
                "authenticationRequired": true
            }
        },
        {
```



```
            "route": "/api/v1/data/sales",
            "dataset": "sales",
            "accessControl": {
                "roles": ["admin"],
                "authenticationRequired": true
            }
        }
    ]
}
```

Figure 103

In this example, `routes.json` defines three routes, each pointing to a data set specified in `storage.json`:

- Users data set
  - Route: `/api/v1/data/users`
  - Dataset: `users` (refers to the `users` key in `storage.json`)
  - Access control: Only accessible by users with `admin` or `user` roles, and authentication is required.
- Products data set
  - Route: `/api/v1/data/products`
  - Dataset: `products` (refers to the `products` key in `storage.json`)
  - Access control: Only accessible by users with the `admin` role, and authentication is required.
- Sales data set
  - Route: `/api/v1/data/sales`
  - Dataset: `sales` (refers to the `sales` key in `storage.json`)
  - Access control: Only accessible by users with the `admin` role, and authentication is required.

Table 5 — Access Control Attributes

Attribute	Description
Roles	Specifies the roles that are allowed to access the data set.
Authentication required	Specifies whether authentication is required to access the data set.

Access control ensures that data can be securely and efficiently retrieved by authorized users.

Conformance

A Capsium package / packager / reactor conforming to class `authentication` SHALL satisfy requirements authentication-1 to authentication-6 of this clause. The conformance class is defined in [conformance-authentication].

Module: Handler routes

The handler routes module extends resource routing with dynamic HTTP API routes: routes that map a URL path and an HTTP method to an executable handler file contained in the package. Where the core package serves only static resources and datasets, a package claiming this module can ship API endpoints whose responses are produced by code.

A reactor claiming this module executes the handler of a matching route. A reactor that does not claim this module shall answer requests to handler routes with `501 Not Implemented` (see the reactor annexes for examples of this behavior).

Additional requirements

1) A handler route in `routes.json` shall declare a `path`, an HTTP `method`, and a `handler` that references a script file contained in the package.
2) The `handler` path shall be package-relative and shall resolve to an existing file in the package.

3) A reactor claiming this module shall, on a request whose path and method match a handler route, execute the referenced handler and deliver its response to the client. The handler shall be an ECMAScript module whose default export — or whose named `fetch` export — is a function of the shape `(request: Request) => Response | Promise<Response>`.
4) Requests to a handler route with an unsupported HTTP method shall be answered with `405 Method Not Allowed` and an `Allow` header listing the supported methods.
5) Header responses declared for the route — inline in `routes.json` or through external header files (Clause 5.7.5) — shall be applied to the handler's response.
6) A failure to import the handler module, or a runtime failure during its execution, shall be answered with `502 Bad Gateway`.
7) A reactor without a JavaScript execution contract shall answer handler routes with `501 Not Implemented`.

HTTP API routes in `routes.json`

The core `routes.json` file (Clause 5, Resource routing) maps URL paths to resources; this module adds route entries that carry a `method` and a `handler` instead of a `resource`.

Example `routes.json` with HTTP API routes

```
{
    "index": "index.html",
    "routes": [
        {
            "path": "/api/v1/users",
            "method": "GET",
            "handler": "api/v1/users/getUsers.js"
        },
        {
            "path": "/api/v1/users",
            "method": "POST",
            "handler": "api/v1/users/createUser.js"
        },
        {
            "path": "/api/v1/users/:id",
            "method": "PUT",
            "handler": "api/v1/users/updateUser.js"
        },
        {
            "path": "/api/v1/users/:id",
            "method": "DELETE",
            "handler": "api/v1/users/deleteUser.js"
        }
    ]
}
```

Figure 104

In this example, the `routes.json` file defines dynamic HTTP API endpoints: each route binds a path (with optional path parameters such as `:id`) and an HTTP method to a handler script in the package, which is executed to produce the response.

Handler routes participate in route visibility (`exported` / `private`, Clause 5.7.6) like any other route, and can be inherited and processed by dependent packages under the composite packages module ([module-composite]); request headers supplanted with `requestHeaders` are applied when the request is forwarded to the handler.

JavaScript handler contract

A handler is an ECMAScript module loaded from the package content. The reactor imports the module and invokes its handler function for each matching request:

```
// content/api/v1/hello.js
export default async function (request) {
  return new Response(JSON.stringify({ hello: "world" }), {
    status: 200,
    headers: { "Content-Type": "application/json" }
  });
}

// alternatively, a named "fetch" export:
export function fetch(request) {
  return new Response("OK");
}
```

Figure 105

The handler function receives a `Request` and returns a `Response` or a `Promise` resolving to a `Response`, following the fetch handler contract of the web platform. The reactor matches the route's declared HTTP `method` before dispatching; a mismatch is answered `405 Method Not Allowed` with an `Allow` header. A module that fails to import, or a handler that throws or rejects, is answered `502 Bad Gateway`.

Sandboxing guidance	Handlers are package-supplied code and should be executed with least privilege. Browser-based reactors should execute handlers inside a service worker or an equivalent isolated browsing context, with a restrictive Content Security Policy (for example, `default-src 'self'`, no `unsafe-eval`) so that a handler can only reach the resources of its own package. A reactor without such a JavaScript execution contract shall not attempt to run handlers and shall answer `501 Not Implemented`.

Conformance

A Capsium package / packager / reactor conforming to class `handler-routes` SHALL satisfy requirements handler-routes-1 to handler-routes-7 of this clause. The conformance class is defined in [conformance-handler-routes].

Module: Testing

The testing module defines a YAML-based domain-specific language (DSL) for describing tests to be executed against Capsium packages. The DSL is designed to be programming language-independent, allowing for implementations in various languages, and covers HTTP routes, file existence, data validation, and configuration testing.

Testing is optional: a conformant core package is not required to carry tests. A package, packager, or test runner that claims the `testing` module shall satisfy the additional requirements of this clause.

Additional requirements

1) A Capsium test file shall consist of a top-level `tests` list, each entry declaring a `name` and a `type`.
2) A test of type `route` shall declare `url` and `expected_status`, and may declare `response_contains`.

3) A test of type `file` shall declare `path`.
4) A test of type `data_validation` shall declare `format`, `data_file`, and `schema_file`.
5) A test of type `config` shall declare `format` and `config_file`.
6) A test runner claiming this module shall correctly interpret and execute the tests of all four types defined in this clause.

YAML structure

The Capsium testing YAML structure consists of a list of tests, each defined with specific attributes depending on the type of test. The top-level structure is as follows:

```
tests:
  - name: <Test Name>
    type: <Test Type>
    ...
```

Figure 106

Test types and options

Route testing

Route testing is used to verify the behavior of HTTP endpoints. Each route test includes the following attributes:

```
- name: <Test Name>
  type: route
  url: <URL>
  expected_status: <HTTP Status Code>
  response_contains: <Optional String to Check in Response>
```

Figure 107

— `name`: A descriptive name for the test.
— `type`: Must be `route` for route testing.
— `url`: The URL of the HTTP endpoint to be tested.
— `expected_status`: The expected HTTP status code (e.g., `200`).
— `response_contains`: (Optional) A string that should be present in the response body.

Example:

```
- name: Home Route Test
  type: route
  url: "http://localhost:8000/home"
  expected_status: 200
  response_contains: "Welcome"
```

Figure 108

File testing

File testing checks for the existence of required files. Each file test includes the following attributes:

```
- name: <Test Name>
  type: file
  path: <File Path>
```

Figure 109

— `name`: A descriptive name for the test.

— `type`: Must be `file` for file existence testing.
— `path`: The file path to be checked.

Example:

```
- name: Config File Exists
  type: file
  path: "/path/to/config.json"
```

Figure 110

Data validation

Data validation tests validate datasets against predefined schemas. Each data validation test includes the following attributes:

```
- name: <Test Name>
  type: data_validation
  format: <Data Format>
  data_file: <Data File Path>
  schema_file: <Schema File Path>
```

Figure 111

— `name`: A descriptive name for the test.
— `type`: Must be `data_validation` for data validation tests.
— `format`: The format of the data file (e.g., `json`, `yaml`).
— `data_file`: The path to the data file to be validated.
— `schema_file`: The path to the schema file to validate against.

Example:

```
- name: JSON Data Validation
  type: data_validation
  format: json
  data_file: "/path/to/datafile.json"
  schema_file: "/path/to/schemafile.json"
```

Figure 112

Configuration testing

Configuration testing ensures that configuration files are correctly structured and valid. Each configuration test includes the following attributes:

```
- name: <Test Name>
  type: config
  format: <Config Format>
  config_file: <Config File Path>
```

Figure 113

— `name`: A descriptive name for the test.
— `type`: Must be `config` for configuration file validation.
— `format`: The format of the configuration file (e.g., `json`, `yaml`).
— `config_file`: The path to the configuration file to be validated.

Example:

```
- name: JSON Config Validation
  type: config
```



```
  format: json
  config_file: "/path/to/config.json"
```

Figure 114

Complete example

Below is a complete example of a Capsium package testing YAML file, demonstrating various test types and their options:

```
tests:
  - name: Home Route Test
    type: route
    url: "http://localhost:8000/home"
    expected_status: 200
    response_contains: "Welcome"

  - name: API Route Test
    type: route
    url: "http://localhost:8000/api/data"
    expected_status: 200
    response_contains: "data"

  - name: Config File Exists
    type: file
    path: "/path/to/config.json"

  - name: Data File Exists
    type: file
    path: "/path/to/datafile.json"

  - name: JSON Data Validation
    type: data_validation
    format: json
    data_file: "/path/to/datafile.json"
    schema_file: "/path/to/schemafile.json"

  - name: YAML Data Validation
    type: data_validation
    format: yaml
    data_file: "/path/to/datafile.yaml"
    schema_file: "/path/to/schemafile.yaml"

  - name: JSON Config Validation
    type: config
    format: json
    config_file: "/path/to/config.json"
```

Figure 115

Conformance

To conform to this module, an implementation must correctly interpret and execute the tests defined in the Capsium package testing YAML format as described in this clause. Implementations may be developed in any programming language, provided they adhere to the specified structure and options.

A Capsium package / packager / test runner conforming to class `testing` SHALL satisfy requirements testing-1 to testing-6 of this clause. The conformance class is defined in [conformance-testing].

Module: Registries

The registries module defines the static Capsium registry: a distribution point from which Capsium packages are published, resolved, and installed. A registry is a plain directory or a static HTTPS endpoint — it requires no server-side logic beyond file serving — so packages can be distributed from any web server, object store, or filesystem directory.

Registries are optional: a conformant core package, packager, or reactor works without them, activating packages from local `.cap` files or package directories. A package, packager, or reactor that claims the `registries` module shall satisfy the additional requirements of this clause.

Additional requirements

1) A Capsium registry shall be a directory or an HTTPS base URL containing an `index.json` file at its root, structured as specified in this clause ([registry-index]).
2) Each package version listed in the index shall be stored as a `.cap` file at a path relative to the registry root, with the SHA-256 checksum and size in bytes recorded in the index entry.
3) When a registry is accessed over the network, HTTPS shall be used; plain HTTP shall only be accepted for loopback addresses (`localhost`, `127.0.0.0/8`, `::1`).
4) A registry write (push) shall validate the package, copy the `.cap` file into the registry, recompute the SHA-256 checksum and size of the copied file, and rewrite `index.json` atomically (write to a temporary file, then rename) so that readers never observe a partially written index.
5) Resolution of a package reference against a registry shall select the newest indexed version that satisfies the requested semantic version range (Clause 5.3.4).
6) Installation from a registry shall download the selected `.cap` file, verify its SHA-256 checksum against the index entry, reject the download on mismatch, and install the verified file into a package store under the name `<name>-<version>.cap`, updating the store index ([module-composite]).
7) Mounts and dependencies may reference a package by a `capsium://` URI of its GUID; such references shall be resolved through the resolution chain *bundle → store → registry* defined by the composite packages module ([module-composite]).

Registry layout

A registry root contains an `index.json` file mapping the GUID of every published package to its name and to the set of published versions. Each version entry records the `.cap` file path (relative to the registry root), its SHA-256 checksum, and its size in bytes:

```
{
  "packages": {
    "capsium://example.com/hello-world": {
      "name": "hello-world",
      "versions": {
        "1.0.0": {
          "file": "hello-world-1.0.0.cap",
          "sha256": "9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f0
0a08",
          "size": 20480
        },
        "1.1.0": {
          "file": "hello-world-1.1.0.cap",
          "sha256": "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852
b855",
          "size": 21504
```



```
          }
        }
      }
    }
}
```

Figure 116

Requirements and specifications:

1) The `packages` object shall be keyed by the package GUID in URI format (Clause 5, Identifier).
2) Each package entry shall carry the package `name` from `metadata.json` and a `versions` object keyed by the semantic version string.
3) Each version entry shall carry:
   `file` the path of the `.cap` file, relative to the registry root;
   `sha256` the SHA-256 checksum of the `.cap` file, as a lowercase hexadecimal string;
   `size` the size of the `.cap` file in bytes, as an integer.
4) Registry clients shall treat `index.json` as the single source of truth for the contents of the registry: a `.cap` file present in the registry but absent from the index is not installable.

Because a registry is static content, it can be served by any web server or object store capable of serving files over HTTPS, or used directly as a filesystem directory. A registry served over HTTPS is read-only: the push operation applies to filesystem directory registries.

Registry operations

Push

Publishing a package to a registry (`push`) shall:

1) Validate the package as specified in [management-clause] (Validation); an invalid package shall not be published.
2) Copy the `.cap` file into the registry at its index-recorded path.
3) Recompute the SHA-256 checksum and size of the copied file (rather than trusting caller-supplied values).
4) Rewrite `index.json` atomically — serialize the updated index to a temporary file in the same directory, then rename it over `index.json` — so that concurrent readers always see either the previous or the updated index, never a partial one.

Resolve

Resolution maps a package GUID and a semantic version range to a concrete version entry of the index:

1) The resolver shall consider every version indexed under the GUID.
2) Among the versions satisfying the requested range (Clause 5.3.4), the resolver shall select the newest (highest) version.
3) When no indexed version satisfies the range, resolution shall fail and the failure shall be reported to the caller.

Install

Installation transfers a resolved package version from a registry into a local package store ([module-composite]):

1) The installer shall download the `.cap` file recorded in the resolved index entry.
2) The installer shall verify the SHA-256 checksum of the downloaded file against the `sha256` of the index entry and shall reject the download on mismatch; a rejected file shall not be installed.

3) The verified file shall be installed into the store as `<name>-<version>.cap`, and the store index shall be updated accordingly.

Referencing packages from a registry

A package stored in a registry is referenced by the `capsium://` URI of its GUID, both in dependency declarations (Clause 5.3.4) and in reactor mount maps ([reactor-clause]). Resolution of such references follows the *bundle → store → registry* chain of the composite packages module ([module-composite]): a reference is first looked up among the packages bundled with the referring package, then in the local package store, and finally installed from a configured registry.

Command-line interface (informative)

The following commands of the `capsium` CLI (Annex A) illustrate the registry operations:

```
capsium package push example-package-1.0.0.cap --registry /srv/registry
capsium install capsium://example.com/hello-world \
    --constraint ">=1.0.0" --registry https://registry.example.com \
    --store ~/.capsium/store
capsium reactor serve capsium://example.com/hello-world
```

Figure 117

Conformance

A Capsium package / packager / reactor conforming to class `registries` SHALL satisfy requirements registries-1 to registries-7 of this clause. The conformance class is defined in [conformance-registries].

Module: Writable packages

The writable packages module makes an activated Capsium package interactive: while the base package remains immutable, a reactor claiming this module accepts writes — content changes, dataset record creation, update, and deletion — records them in a reactor-side overlay, and serves the merged result. It also defines the GraphQL API over datasets and the operation that folds the accumulated changes back into a new, valid `.cap` artifact.

Writability is optional: a conformant core package is immutable and a core reactor serves it read-only. A package, packager, or reactor that claims the `writable-packages` module shall satisfy the additional requirements of this clause. The overlay mechanics this module builds on — layers, merged resolution, and tombstones — are defined by the layered storage module ([module-layered-storage]); the folding of overlays into a new package follows the composite packages module ([module-composite]).

Additional requirements

1) A package whose `metadata.json` declares `readOnly: true` is immutable: a reactor shall answer every write request to it with `403 Forbidden`. When `readOnly` is absent or `false`, the package is writable.
2) The base package shall never be modified. Writes shall go to a new, append-only top layer created and maintained by the reactor (reactor-side), and merged resolution shall follow the top-first order of [module-layered-storage].
3) Deletions shall be recorded as tombstones: a `.capsium-tombstones` file containing a JSON array of content-relative paths, kept in the topmost writable layer ([module-layered-storage]).
4) A reactor claiming this module shall provide the dataset CRUD API of this clause for every JSON-backed dataset of a writable package, with the specified status codes: `201` with a `Location` header and the stored item on creation, `200` on read and update, `204` on deletion, `404` for unknown datasets and items, `405` for unsupported methods, `409` on identifier conflict,

422 with details on schema violation, and 501 for dataset kinds the reactor cannot modify (for example SQLite).

5) Data item identity shall be determined by an id field of the item when present, and by the 1-based index of the item within the dataset, expressed as a string, otherwise.
6) A reactor claiming this module shall accept content writes: PUT of a content path creates or overwrites the corresponding content in the top layer (routes resolve on demand), and DELETE of a content path records a tombstone.
7) A reactor claiming this module should provide the GraphQL API of this clause over the datasets of a writable package; failures caused by the client request shall be reported in the GraphQL errors array and shall not produce HTTP 500 responses.
8) Writes shall become visible on the next request without a reactor restart (hot-swap), and the overlay state shall persist in the reactor work directory and be reloaded on restart.
9) A reactor claiming this module shall provide the save operation of this clause, folding base package and overlays into a new .cap file that shall pass package validation before it is reported.

Write gate: readOnly

The readOnly key of metadata.json (Clause 5.3.6) decides whether an activated package accepts writes:

— readOnly: true — the package is immutable; every write request (POST, PUT, DELETE on content or data endpoints, mutations over GraphQL) is answered 403 Forbidden.
— readOnly absent or false — the package is writable through the APIs of this clause.

Overlay write model

The base package is never modified. On the first write to a writable package, the reactor creates a new top layer above all layers of the package ([module-layered-storage]) and maintains it in its work directory:

— The top layer is append-only: every write adds content to it; nothing is ever removed from it or from any lower layer.
— Merged resolution is top-first, so newly written content shadows the content of lower layers.
— Deletions are recorded in the .capsium-tombstones file of the topmost writable layer as a JSON array of content-relative paths. A tombstoned path resolves to 404 thereafter, even when a lower layer still holds the file; writing the path again removes the tombstone.

Dataset CRUD API

For a writable package, each dataset route /api/v1/data/<dataset> (Clause 5.7.3) accepts the following operations. All request and response bodies are application/json.

Table 6 — Dataset CRUD operations of a writable package

Operation	Behavior
POST /api/v1/data/<dataset>	Creates a data item. On success: 201 Created with a Location header identifying the new item and the stored item as the body.
GET /api/v1/data/<dataset>/<id>	Returns the identified item with 200 OK, or 404 Not Found when the dataset or the item does not exist.
PUT /api/v1/data/<dataset>/<id>	Replaces the identified item. On success: 200 OK with the updated item. 404 Not Found when the dataset or item is absent; 422 Unprocessable Content when the item violates the dataset schema, with a body detailing the violations.
DELETE /api/v1/data/<dataset>/<id>	Deletes the identified item. On success: 204 No Content; 404 Not Found when the dataset or the item does not exist.
Item identity	An item is identified by the value of its id field when the item carries one; otherwise items are identified by their 1-based position within the dataset, expressed as a string ("1", "2", ...).

Error reporting	The API uses the following status codes consistently: — `400 Bad Request` — the request body is not well-formed JSON; — `403 Forbidden` — the package is read-only; — `404 Not Found` — the dataset or item does not exist; — `405 Method Not Allowed` — the endpoint does not support the request method; — `409 Conflict` — the item identifier is in conflict: it already exists on creation, or the body identifier does not match the path on update; — `422 Unprocessable Content` — the item violates the dataset schema; the response body details the violations; — `501 Not Implemented` — the dataset kind cannot be modified by this reactor (for example, SQLite datasets).

Content writes

Beyond datasets, a reactor claiming this module accepts writes to the content of a writable package:

— `PUT <path>` creates or overwrites the content served at `<path>`: the body is written to the top layer, an already routed resource is replaced in the overlay, and previously unrouted paths become routable on demand — a subsequent `GET <path>` serves the written content.
— `DELETE <path>` records a tombstone for the content served at `<path>`. The path resolves to `404 Not Found` thereafter, even when a lower layer still holds the file; `DELETE` of a path that has no servable content is answered `404 Not Found`.
— A subsequent `PUT <path>` of a tombstoned path removes the tombstone and serves the newly written content.

GraphQL API

A reactor claiming this module should expose a GraphQL endpoint at `<mount>/graphql`, accepting both `POST` and `GET` requests, over the datasets of a writable package:

— For each dataset, the schema provides a list query field (named after the dataset) with an optional `id:` argument selecting a single item.
— For each dataset, the schema provides `create<Dataset>`, `update<Dataset>`, and `delete<Dataset>` mutations.
— Item types are derived from the dataset's JSON schema when one is declared (Clause 5.7.3); otherwise items are typed as a permissive JSON scalar. Datasets that cannot be represented as a JSON collection (for example, SQLite datasets) are omitted from the schema.
— Failures caused by the client request — validation errors, unknown items, conflicts — are reported in the `errors` array of the GraphQL response; they shall not be reported as HTTP `500` responses.

Hot-swap and persistence

Writes take effect immediately: a change is visible on the next request without restarting the reactor. The overlay state — the top layer content and its tombstones — persists in the reactor work directory, so a reactor restart reloads the accumulated changes and serving continues from the merged state.

Saving a writable package

The save operation folds the current merged state of a writable package back into a self-contained artifact:

`POST /package/`	builds a new package from the base package plus the accumulated overlays: tombstoned paths are removed, added and overwritten content

`<name>/save` — is incorporated, recorded dataset changes are replayed into the dataset sources, and `manifest.json`, `routes.json`, and `security.json` are regenerated for the result. The new package is written as `<name>-<version>.cap` with the patch component of the version incremented, and it shall pass package validation ([management-clause], Validation) before completion; the response returns the path of the new `.cap` file and its SHA-256 checksum, with the package name and new version. Signing artifacts are not folded into the saved package: it is unsigned until re-signed with [module-signatures].

Conformance

A Capsium package / packager / reactor conforming to class `writable-packages` SHALL satisfy requirements writable-packages-1 to writable-packages-9 of this clause. The conformance class is defined in [conformance-writable-packages].

Capsium Reactor

The Capsium reactor is the execution environment responsible for running Capsium packages. It supports various deployment environments, provides APIs for introspection, manages user authentication and data security, and ensures reliable and trusted execution of packages.

Structure

The structure of a Capsium reactor typically includes:

Core Engine — The core runtime that executes Capsium packages.

Configuration Files — Settings and configurations for the reactor's operation.

Plugins — Optional plugins for additional functionality (e.g., authentication, logging).

APIs — Interfaces for introspection, monitoring, and management.

Directory structure example: ` capsium-reactor/ ├── core/ │ ├── engine.js │ ├── config/ │ │ ├── settings.json │ │ └── plugins.json ├── plugins/ │ ├── auth-plugin.js │ ├── logging-plugin.js ├── apis/ │ ├── introspection-api.js │ ├── monitoring-api.js │ └── package-api.js ├── logs/ └── data/ `

Operation Environments

The Capsium reactor is designed to operate in various environments to provide flexibility and scalability.

Reactor in the Browser Natively or as a Plugin

Natively — The reactor can run directly in the browser using WebAssembly or JavaScript, allowing for client-side execution of Capsium packages.

- Requirements: Modern web browser with WebAssembly and JavaScript support.
- Use Cases: Client-side applications, browser extensions.

As a Plugin — The reactor can be embedded as a browser plugin, providing additional capabilities and tighter integration with browser features.

- Requirements: Plugin installation, browser compatibility.
- Use Cases: Enhanced browser extensions, specific web application functionalities.

Reactor in the Web Server as a Plugin

Web Server Plugin	The reactor can be deployed as a plugin in web servers such as Apache, Nginx, or Node.js. — Requirements: Compatible web server, plugin installation. — Use Cases: Server-side applications, API backends.

Example configuration for Node.js: `javascript const capsiumReactor = require('capsium-reactor'); const express = require('express'); const app = express();

app.use('/capsium', capsiumReactor());

app.listen(3000, () ⇒ { console.log('Capsium reactor running on port 3000'); }); `

Reactor on Cloud Services (AWS S3 or GitHub Pages)

AWS S3	— Deployments can be hosted on AWS S3 as static websites. — Requirements: AWS S3 bucket, configuration for static website hosting. — Use Cases: Hosting static applications, deploying packages with minimal backend requirements.
GitHub Pages	— Deployments can be hosted on GitHub Pages for easy access and version control. — Requirements: GitHub repository, Pages configuration. — Use Cases: Open-source projects, documentation sites.

Example GitHub Pages setup: ` 1. Create a GitHub repository. 2. Push your Capsium package to the repository. 3. Enable GitHub Pages in the repository settings. 4. Access your package at https://<username>.github.io/<repository>/. `

Multiple mounted packages

A reactor may mount several Capsium packages at once. Each mount is a map carrying the mount `path` and the package `source` (a `.cap` file, a package directory, or a `capsium://` package reference resolved through [module-composite] and [module-registries]). When mount paths are not given explicitly, the first package is mounted at `/` and every further package at `/<metadata.name>/`. Mount paths shall be unique within a reactor; conflicting mounts are rejected.

Dispatch	Requests are dispatched to the mounted package with the longest matching mount path prefix.
Introspection	The introspection endpoints of this clause aggregate all mounted packages.
Per-package endpoints	Endpoints under `/package/` resolve their `<name>` path segment against the `metadata.name` of every mounted package; an unknown name is answered `404 Not Found`.

Serving rules

A conformant reactor serves package routes with the following rules.

Methods
: Package routes are served for `GET` and `HEAD` requests only. Any other method on a package route is answered `405 Method Not Allowed` with an `Allow` header listing the supported methods. A `HEAD` request returns the same headers as the corresponding `GET` response, without a response body.

Caching
: Unless a route or mount declares its own `Cache-Control`, static resources are served with `Cache-Control: public, max-age=31536000` (one year); the default may be overridden per route (Clause 5.7.5) or per mount.

Media types
: Static resources are served with the MIME type recorded in `manifest.json`; when the manifest has no entry for a resource, the reactor falls back to a media type derived from the file name extension.

Datasets
: Dataset routes respond with `application/json`. JSON datasets are served natively; YAML datasets are parsed and served as JSON; CSV datasets may be served as JSON. SQLite datasets are optional: a reactor that does not support them answers their routes with `501 Not Implemented`.

Handler routes
: Routes that bind a method and path to an executable handler are executed only by reactors with a JavaScript execution contract; any other reactor answers them with `501 Not Implemented` ([module-handler-routes]).

Write requests
: Write requests (`PUT`, `POST`, `DELETE` on content and dataset endpoints) are served only by reactors claiming the writable packages module ([module-writable-packages]).

HTTP API for Introspection of Reactor

The reactor provides an HTTP API for introspection, allowing users to query the reactor's status, configuration, and operational metrics.

Endpoints
: — `/introspect/status`: Returns the current status of the reactor.
— `/introspect/config`: Returns the reactor's configuration details.
— `/introspect/metrics`: Returns operational metrics (e.g., uptime, resource usage).

Response shapes
: The reactor-level introspection endpoints answer `GET` requests with `application/json` bodies of the following shapes.

`/introspect/status`

```
{
  "status": "running",
  "uptime": 3600,
  "packagesLoaded": 5
}
```

Figure 118

`uptime` is the reactor uptime in seconds; `packagesLoaded` is the number of mounted packages.

`/introspect/config`
: The configuration representation is reactor-defined, but the endpoint shall redact secrets: credentials such as OAuth secrets, private key paths, and the userinfo component of configured URLs shall never appear in the response.

`/introspect/metrics`

```
{
  "uptime": 3600,
```



```
  "requestsTotal": 12480,
  "requestsByStatus": {
    "200": 12001,
    "404": 475,
    "500": 4
  }
}
```

Figure 119

`requestsTotal` counts all served requests; `requestsByStatus` counts them per HTTP status code.

HTTP API for Introspection of Package

The reactor also provides an HTTP API to introspect individual Capsium packages, enabling users to retrieve package-specific information. The `<name>` path segment is resolved against the `metadata.name` of every mounted package; an unknown name is answered `404 Not Found`.

Endpoints
- `/package/<name>/status`: Returns the status of the specified package.
- `/package/<name>/metadata`: Returns the metadata of the specified package.
- `/package/<name>/logs`: Returns the logs related to the specified package. The optional `lines` query parameter limits the number of log lines returned (default 100).

Example API response for `/package/<name>/metadata`:

```
{
  "name": "example-package",
  "version": "1.0.0",
  "description": "An example Capsium package",
  "author": "Author Name",
  "guid": "capsium://example.com/example-package"
}
```

Figure 120

A reactor claiming the writable packages module additionally answers `POST /package/<name>/save`, folding the package's accumulated overlay changes into a new `.cap` artifact ([module-writable-packages]).

Access to Activated Capsium Package Information, Metadata

The reactor maintains detailed information and metadata for each activated Capsium package. This information includes version details, dependencies, and configuration settings.

Access Methods
- Via HTTP API: Endpoints such as `/package/<name>/metadata` provide access to package metadata.
- Via Configuration Files: Metadata can be stored and accessed through configuration files within the reactor's directory structure.

Example metadata structure:

```
{
  "packages": {
    "example-package": {
      "name": "example-package",
      "version": "1.0.0",
      "description": "An example Capsium package",
      "author": "Author Name",
```

```
      "dependencies": ["dependency1", "dependency2"],
      "config": {
        "option1": "value1",
        "option2": "value2"
      }
    }
  }
}
```

Figure 121

Monitoring and Logging

The Capsium reactor includes robust monitoring and logging capabilities to ensure smooth operation and facilitate troubleshooting.

Monitoring
- Health Checks: Periodic checks to ensure the reactor and its packages are running correctly.
- Metrics Collection: Collection of performance metrics such as CPU and memory usage, request counts, and error rates.

Logging
- Access Logs: Logs of all incoming requests and their responses.
- Error Logs: Detailed logs of errors encountered during operation.
- Custom Logs: Logs generated by individual packages for specific events.

Example logging configuration in `plugins/logging-plugin.js`: `javascript const fs = require('fs'); const path = require('path');

module.exports = function loggingPlugin(req, res, next) { const logEntry = `${new Date().toISOString()} - ${req.method} ${req.url}\n`; fs.appendFileSync(path.join(__dirname, '../logs/access.log'), logEntry); next(); }; `

Handling User Authentication (Apache passwd, External OAuth Authentication Defined by Packages)

The reactor supports various user authentication methods to secure access to its resources. The package-side configuration of these methods, and the access control of dataset routes, are defined by the authentication module ([module-authentication]); a reactor implements this behavior when it claims the Capsium authentication class ([conformance-authentication]).

Apache passwd — Uses `.htpasswd` files for basic HTTP authentication.
- Requirements: `.htpasswd` file containing user credentials.
- Use Cases: Simple authentication for small deployments.

External OAuth — Supports OAuth authentication as defined by individual packages.
- Configuration: OAuth provider details need to be configured.
- Use Cases: Integration with third-party authentication providers like Google, Facebook, or custom OAuth servers.

Example configuration for OAuth in `auth-plugin.js`: `javascript const passport = require('passport'); const OAuth2Strategy = require('passport-oauth2').Strategy;

passport.use(new OAuth2Strategy({ authorizationURL: 'https://example.com/oauth/authorize', tokenURL: 'https://example.com/oauth/token', clientID: 'your-client-id', clientSecret: 'your-client-secret', callbackURL: 'https://your-app.com/callback' }, (accessToken, refreshToken, profile, cb) ⇒ { // Verify and handle user profile cb(null, profile); }));

app.use(passport.initialize()); app.get('/auth/example', passport.authenticate('oauth2')); app.get('/callback', passport.authenticate('oauth2', { failureRedirect: '/' }), (req, res) ⇒ { res.redirect('/'); }); `

Decrypting User Data

The reactor includes mechanisms for securely decrypting user data, ensuring it remains protected while in transit and at rest. The package-side encryption configuration is defined by the encryption module ([module-encryption]); a reactor implements this behavior when it claims the Capsium encryption class ([conformance-encryption]).

Encryption Algorithms	Supports standard encryption algorithms such as AES-256.
Key Management	Secure storage and management of encryption keys.
API for Decryption	Provides an API for decrypting user data when needed.

Example decryption function in `data-handler.js`: `javascript const crypto = require('crypto');

function decryptData(encryptedData, key) { const decipher = crypto.createDecipher('aes-256-cbc', key); let decrypted = decipher.update(encryptedData, 'hex', 'utf8'); decrypted += decipher.final('utf8'); return decrypted; } `

Updating Modifiable Capsium Packages

A Capsium package is immutable as distributed. Modifications of an activated package — content writes, dataset changes, and deletions — are defined by the writable packages module ([module-writable-packages]): writes accumulate in a reactor-side overlay, are visible on the next request (hot-swap), persist across restarts, and can be folded into a new package version with the save operation (`POST /package/<name>/save`). A reactor implements this behavior when it claims the Capsium writable-packages class ([conformance-writable-packages]).

Update Mechanism	Supports hot-swapping of packages with minimal disruption.
Version Control	Keeps track of package versions and allows rollback if necessary.
API for Updates	Provides an API for updating packages.

Trusted Execution

The reactor ensures trusted execution of Capsium packages by enforcing security measures and maintaining integrity throughout the runtime.

Sandboxing — Each package runs in an isolated environment to prevent interference and enhance security.

- Requirements: Use of technologies like Docker containers or virtual machines.
- Use Cases: Running untrusted code, ensuring package isolation.

Code Signing — All packages must be signed by a trusted authority to verify their integrity and authenticity.

- Requirements: Digital certificates and a trusted certificate authority (CA).
- Use Cases: Preventing tampering and ensuring only authorized packages are executed.

Integrity Checks	Regular integrity checks are performed to ensure that packages have not been altered. — Methods: Hash verification, signature validation. — Use Cases: Detecting unauthorized changes, maintaining trust.
Audit Logs	Detailed logs of all operations and accesses are maintained to provide an audit trail. — Requirements: Comprehensive logging infrastructure. — Use Cases: Security audits, forensic analysis.

Example sandboxing setup using Docker: `dockerfile # Dockerfile for a Capsium package FROM node:14

WORKDIR /app

COPY . .

RUN npm install

CMD [“node”, “index.js”] `

Example code signing process: . Generate a key pair: `bash openssl genrsa -out private.key 2048 openssl rsa -in private.key -pubout -out public.key `

1) Sign the package: `bash openssl dgst -sha256 -sign private.key -out package.sig package.zip `
2) Verify the signature: `bash openssl dgst -sha256 -verify public.key -signature package.sig package.zip `

With these detailed requirements and specifications, the Capsium reactor ensures a secure, flexible, and robust environment for executing Capsium packages across various deployment scenarios.

Monitoring HTTP API

The Capsium reactor provides a comprehensive Monitoring HTTP API designed to expose various details about the reactor and its packages. This API is particularly useful for browser users who need to access metadata, routes, content hashes, and validity information directly from the browser. All endpoints are namespaced under `/api/v1/introspect`.

Exposing metadata.json

The `metadata.json` file contains essential information about the Capsium packages, including their names, versions, authors, and descriptions. The Monitoring HTTP API provides an endpoint to retrieve this information.

Endpoint	`/api/v1/introspect/metadata`
Method	GET
Response	JSON containing the metadata of all active packages.

Example response:

```
{
  "packages": [
    {
      "name": "example-package",
```



```
      "version": "1.0.0",
      "author": "Author Name",
      "description": "An example Capsium package"
    },
    {
      "name": "another-package",
      "version": "2.0.0",
      "author": "Another Author",
      "description": "Another example Capsium package"
    }
  ]
}
```

Figure 122

Exposing routes.json

The `routes.json` file lists all the available routes provided by the Capsium packages. This is critical for understanding the API surface and available endpoints.

Endpoint	`/api/v1/introspect/routes`
Method	GET
Response	JSON containing the routes of all active packages.

Example response:

```
{
  "routes": [
    {
      "package": "example-package",
      "routes": [
        {
          "method": "GET",
          "path": "/example"
        }
      ]
    },
    {
      "package": "another-package",
      "routes": [
        {
          "method": "POST",
          "path": "/another"
        }
      ]
    }
  ]
}
```

Figure 123

Exposing packaged content hashes

To ensure content integrity, the reactor can expose the hashes of the packaged content. This allows users to verify that the content has not been tampered with.

Endpoint	`/api/v1/introspect/content-hashes`

Method GET

Response JSON containing the hashes of all packaged content.

Example response:

```
{
  "contentHashes": [
    {
      "package": "example-package",
      "hash": "abcd1234efgh5678ijkl9012mnop3456qrst6789uvwx0123yzab4567cdef8901"
    },
    {
      "package": "another-package",
      "hash": "1234abcd5678efgh9012ijkl3456mnop6789qrst0123uvwx4567yzab8901cdef"
    }
  ]
}
```

Figure 124

Exposing content validity information

The reactor can also expose information regarding the validity of the packaged content. This includes checks on whether the content has passed integrity checks and is trusted for execution.

Endpoint `/api/v1/introspect/content-validity`

Method GET

Response JSON containing the validity status of all packaged content. Each entry carries the package name, the validity outcome, and the time of the check, with the reason when invalid; entries may additionally carry `signed` and `encrypted` flags and, for signed packages, the `signatureValid` outcome of signature verification ([module-signatures], [module-encryption]).

Example response:

```
{
  "contentValidity": [
    {
      "package": "example-package",
      "valid": true,
      "lastChecked": "2024-05-28T12:34:56Z",
      "signed": true,
      "encrypted": false,
      "signatureValid": true
    },
    {
      "package": "another-package",
      "valid": false,
      "lastChecked": "2024-05-28T12:34:56Z",
      "signed": false,
      "encrypted": false,
      "reason": "Signature mismatch"
    }
  ]
}
```

Figure 125

Deploy configuration

When deploying a Capsium package to a Capsium reactor, an optional configuration file named `deploy.json` can be provided to control the behavior of the package in the deployed environment. This file allows fine-tuning of various aspects, including logging and monitoring options, data storage, deployment performance requirements, and server-side secrets.

Structure of `deploy.json`

The `deploy.json` file should be a JSON-formatted file with the following sections:

logging	Options for controlling logging behavior.
monitoring	Configuration for monitoring the package.
dataStorage	Settings for data storage.
performance	Deployment performance requirements.
secrets	Server-side secrets, such as OAuth secrets.

Example `deploy.json` File

```
{
  "logging": {
    "level": "DEBUG",
    "file": "/var/log/capsium/example-package.log",
    "format": "json"
  },
  "monitoring": {
    "enabled": true,
    "endpoint": "http://monitoring.example.com/api/v1/metrics",
    "interval": "60s"
  },
  "dataStorage": {
    "type": "filesystem",
    "path": "/var/data/capsium/example-package"
  },
  "performance": {
    "maxMemory": "512MB",
    "maxCPU": "2"
  },
  "secrets": {
    "oauthSecret": "supersecretkey"
  }
}
```

Figure 126

Detailed Specifications

Logging and Monitoring Options

The `logging` and `monitoring` sections control how the package logs information and integrates with monitoring systems.

logging.level	Defines the logging level (e.g., DEBUG, INFO, WARN, ERROR).
logging.file	Specifies the file path where logs should be written.
logging.format	Determines the format of the logs (e.g., plain text, JSON).
monitoring.enabled	A boolean to enable or disable monitoring.
monitoring.endpoint	The URL of the monitoring system's API endpoint.
monitoring.interval	The interval at which monitoring data should be sent.

Data Storage

The `dataStorage` section specifies configurations for data storage.

dataStorage.type	The type of data storage (e.g., filesystem).
dataStorage.path	The file system path where data should be stored.

Deployment Performance Requirements

The `performance` section defines the performance requirements for the deployed package.

performance.maxMemory	The maximum amount of memory the package is allowed to use (e.g., "512MB").
performance.maxCPU	The maximum number of CPU cores the package can utilize (e.g., "2").

Server-Side Secrets

The `secrets` section is used to provide sensitive information such as OAuth secrets.

secrets.oauthSecret	The secret key used for OAuth authentication.

Usage

To deploy a Capsium package with the additional configuration provided in `deploy.json`, include the file during the deployment process.

Example deployment command: `bash capsium deploy example-package@1.0.0 —config deploy.json `

The Capsium reactor will read the `deploy.json` file and apply the specified configurations, ensuring that the package operates according to the defined settings.

Management of Capsium packages

A Capsium package passes through a well-defined lifecycle: it is built from a package directory, inspected, validated, deployed and activated on a Capsium reactor, monitored while serving, and eventually updated or rolled back. This clause specifies the management operations that apply at each stage of that lifecycle and the behavior expected of the tools that perform them.

General

The lifecycle of a Capsium package consists of the following operations:

Building	A package directory conforming to Clause 5 is packed into a single `.cap` file.
Inspecting	The metadata, manifest, routes, and storage configuration of a package are examined without activating the package.
Validation	The package is checked against the requirements of Clause 5 before or after distribution.
Deployment and activation	The package is loaded by a Capsium reactor, which verifies its integrity and serves its routes at a web address.
Monitoring	The reactor exposes the state of the activated package through the Monitoring HTTP API ([reactor-clause]).
Updating and rollback	A new version of the package replaces a deployed one, or a previously deployed version is reinstated.
Dependency resolution	The packages required by a package, as declared in its metadata, are identified and made available.

Management operations are performed by packagers and reactors; the command examples in this clause use the `capsium` command-line interface (Annex A) and the reactor introspection endpoints ([reactor-clause]) to illustrate them.

Building a package

Building assembles a package directory and produces the distributable `.cap` file. The build process shall:

1) Start from a package directory whose structure conforms to Clause 5, containing at minimum a hand-authored `metadata.json`.
2) Generate `manifest.json` from the package content when it is absent, and generate `routes.json` from the manifest when it is absent, applying the generation rules of Clause 5.
3) Generate `security.json` containing a SHA-256 checksum for every file in the package except `security.json` itself (Clause 5, Security).
4) Compress the directory into a single file conforming to the packaging rules of Clause 5, named `{name}-{version}.cap` where `name` and `version` are taken from `metadata.json`.

Example using the `capsium` CLI:

```
capsium package pack -f path-to-package
# => Package created: example-package-1.0.0.cap
```

Figure 127

A built `.cap` file can be unpacked again for inspection or modification:

```
capsium package unpack example-package-1.0.0.cap -o example-package
```

Figure 128

Extraction shall reject archive entries whose paths would be written outside the destination directory (absolute paths, drive letters, `..` segments), so that unpacking an untrusted package cannot overwrite unrelated files.

Inspecting a package

Inspection exposes the configuration of a package without deploying it. Management tooling shall be able to report, for a package directory or a `.cap` file:

— the metadata (`metadata.json`), including name, version, identifiers, and dependencies;
— the manifest (`manifest.json`), generated when absent;
— the routes (`routes.json`), generated when absent;
— the storage configuration (`storage.json`) and its datasets.

Example using the `capsium` CLI:

```
capsium package info     example-package-1.0.0.cap
capsium package metadata example-package-1.0.0.cap
capsium package manifest example-package-1.0.0.cap
capsium package routes   example-package-1.0.0.cap
capsium package storage  example-package-1.0.0.cap
```

Figure 129

Validation

Validation determines whether a package satisfies the requirements of Clause 5. Validation should be performed after building and before distribution, and may be repeated by any recipient of the package.

A validation procedure shall check, and report per check, at least:

metadata — required fields are present and well-formed (kebab-case name, semantic version, URI `guid`, valid `uuid`);

manifest — every declared resource exists in the package;

routes — route targets exist, dataset routes are mounted under `/api/v1/data/`, and the index resolves to an existing HTML file;

storage — dataset sources and schemas exist, and dataset data validates against its declared schema;

security — when `security.json` is present, all checksums match;

content — packaged content does not reference external resources that would break self-containment.

Validation shall produce a per-check report and shall signal failure (for example, through a non-zero process exit status) when any check fails.

Example using the `capsium` CLI:

```
capsium package validate example-package-1.0.0.cap
# => PASS metadata
# => PASS manifest
# => PASS routes
# => PASS storage
# => PASS security
```



```
# => PASS content
```

Figure 130

Deployment and activation on a reactor

Deployment delivers a `.cap` file to a Capsium reactor; activation is the process by which the reactor loads the package and serves its routes at a web address (Clause 4).

On activation the reactor shall:

1) Accept a package from a local `.cap` file or package directory.
2) Verify the integrity of the package against `security.json` when present, and reject a package that fails verification (Clause 5, Security).
3) Load the package configuration, generating `manifest.json` and `routes.json` when absent, and mount every route defined by the package.
4) Serve the package routes as specified in [reactor-clause], and expose the Monitoring HTTP API for the activated package.

An optional deployment configuration file (`deploy.json`, [reactor-clause]) may be provided to tune logging, monitoring, data storage, performance, and server-side secrets for the deployed environment.

Example using the `capsium` CLI:

```
capsium reactor serve example-package-1.0.0.cap --port 8864
# => Starting server on http://localhost:8864
```

Figure 131

Monitoring an activated package

While a package is activated, the reactor answers the Monitoring HTTP API ([reactor-clause]) under `/api/v1/introspect`. Management tooling should use these endpoints to confirm the state of a deployment:

— `GET /api/v1/introspect/metadata` — the metadata of the activated package(s), confirming which name and version are serving;
— `GET /api/v1/introspect/routes` — the routes served per package;
— `GET /api/v1/introspect/content-hashes` — the content hash per package, for comparison against a known-good value;
— `GET /api/v1/introspect/content-validity` — the result of re-verifying the package against `security.json`, including the time of the check and the reason when invalid.

Updating and rollback

Capsium packages are versioned with Semantic Versioning in `metadata.json` (Clause 5), and the built artifact carries the version in its file name (`{name}-{version}.cap`). Versioning enables efficient updates and rollbacks (Clause 4).

Updating
: An update is built as a new `.cap` file whose `version` is incremented according to the nature of the change (Clause 5, Versions). Deploying the new artifact to the reactor replaces the previously activated version.

Rollback
: Because each built `.cap` is immutable and self-contained, rollback is performed by re-deploying the artifact of an earlier version. Management tooling should retain previously deployed artifacts to make rollback possible without rebuilding.

Integrity verification

Integrity verification protects every stage of the lifecycle after building:

— At build time, `security.json` records a SHA-256 checksum for every package file except itself (Clause 5, Security).
— At activation time, the reactor shall re-calculate the checksums and reject the package on any mismatch.
— While serving, verification can be repeated on demand through the `/api/v1/introspect/content-validity` endpoint ([reactor-clause]), which reports the outcome together with a check timestamp.

A package whose content does not match `security.json` shall be rejected; it shall not be served.

Dependency resolution

A package declares its dependencies in `metadata.json` as an object mapping the GUID of each required package to a semantic version range (Clause 5, Dependencies). Dependency resolution is the process of identifying the required packages and versions and making them available so that the dependent package functions correctly.

Resolution shall:

1) Read the `dependencies` object from the package metadata.
2) Select, for each dependency, an available package whose GUID matches and whose version satisfies the declared range.
3) Make the selected packages available to the deployment, either as individually activated packages or composed into a composite package ([module-composite]).

Management tooling should detect unresolvable dependencies — a missing GUID or a version range no available artifact satisfies — and report them before activation rather than serving a partially functional package.

Compliance

This clause defines the conformance classes of the Capsium framework and the requirements that an artifact or implementation shall satisfy to claim conformance to each class. Requirements are expressed as numbered, testable statements and reference the clauses in which the underlying specifications are given.

General

Two kinds of conformance classes are defined.

Core conformance classes — every conformant implementation belongs to one of these:

Core Package	A `.cap` artifact that satisfies the core package requirements of Clause 5.
Core Packager	Software that produces Core Packages.
Core Reactor	Software that activates and serves Core Packages as specified in [reactor-clause].

Module conformance classes — one per Capsium module defined in [modules]: Capsium signatures, Capsium encryption, Capsium layered-storage, Capsium composite, Capsium authentication, Capsium handler-routes, Capsium testing, Capsium registries, and Capsium writable-packages. Module classes are claimed à la carte; an artifact or implementation claiming

a module class shall satisfy every numbered additional requirement of the corresponding module clause, in addition to its core class requirements.

Informative examples of implementations of these classes are given in Annexes A to D.

Core Package

A Capsium package conforms to the Core Package class when all of the following statements are true:

1) The package is distributed as a single compressed file with the `.cap` extension and the MIME type `application/vnd.capsium.package` (Clause 5, General; Packaging options).
2) The package directory structure conforms to the folder hierarchy of Clause 5 (Structure): configuration files reside at the package root, served content under the content directory, and paths inside configuration files are package-relative.
3) The package contains a `metadata.json` file (Clause 5, Metadata file) in which the `name`, `version`, `description`, `guid`, and `uuid` attributes are present, and in which:
   a) `name` is a kebab-case string;
   b) `version` follows Semantic Versioning;
   c) `guid` is a URI;
   d) `uuid` is a valid UUID.
4) When the package declares claimed modules in `metadata.json`, every value of the `modules` key is the identifier of a module defined in [modules], and the package satisfies the package-side requirements of each claimed module.
5) Dependencies, when declared, are expressed in `metadata.json` as an object mapping the GUID of each required package to a semantic version range (Clause 5, Dependencies).
6) The package contains a `manifest.json` file, either provided or generated from the package content according to the generation rules of Clause 5 (Manifest file), and every resource declared in the manifest exists in the package.
7) The package contains a `routes.json` file, either provided or generated from the manifest according to the generation rules of Clause 5 (Resource routing): HTML content is reachable through both its base name and its full file name, and other resources are routed relative to the content path.
8) Every target referenced by `routes.json` exists in the package, and dataset routes are mounted under `/api/v1/data/` (Clause 5, Dataset routes).
9) When the package contains datasets, `storage.json` declares each dataset referenced by a dataset route, and dataset data validates against its declared schema (Clause 5, Storage; Validation).
10) When `security.json` is present, it declares `SHA-256` as the checksum algorithm and lists a SHA-256 checksum for every file in the package except `security.json` itself, and every listed checksum matches the corresponding file (Clause 5, Security).
11) The package is self-contained: served content does not depend on external resources to function (Clause 4, Principles; Clause 5, Validation).

Core Packager

A packager conforms to the Core Packager class when all of the following statements are true:

1) The packager produces packages that satisfy every statement of the Core Package class.
2) When `manifest.json` is absent from the package directory, the packager generates it from the package content according to Clause 5 (Manifest file).
3) When `routes.json` is absent from the package directory, the packager generates it from the manifest according to Clause 5 (Resource routing).
4) The packager writes `security.json` at packaging time, containing a SHA-256 checksum for every file in the package except `security.json` itself (Clause 5, Security).
5) The packager names the built artifact `{name}-{version}.cap`, taking `name` and `version` from `metadata.json` ([management-clause], Building a package).
6) When extracting a `.cap` file, the packager rejects archive entries whose paths would escape the destination directory ([management-clause], Building a package).

7) The packager should offer validation of a package directory or `.cap` file against the Core Package statements, reporting per check and signalling failure when any check fails ([management-clause], Validation).

Core Reactor

A reactor conforms to the Core Reactor class when all of the following statements are true:

1) The reactor activates a package from a `.cap` file or package directory and serves every route defined by the package's `routes.json` as specified in [reactor-clause], including dataset routes under `/api/v1/data/` and the package index route.
2) Package routes are served for `GET` and `HEAD` requests only; `HEAD` returns the response headers without a body, and any other method is answered `405 Method Not Allowed` with an `Allow` header ([reactor-clause], Serving rules).
3) Static resources are served with the MIME types recorded in `manifest.json`, falling back to extension-derived media types, and with `Cache-Control: public, max-age=31536000` unless a route or mount declares its own.
4) Dataset routes respond `application/json`; a reactor that does not support SQLite datasets answers their routes with `501 Not Implemented`.
5) When `security.json` is present in a package, the reactor verifies the package content against the listed SHA-256 checksums on activation (Clause 5, Security; [management-clause], Deployment and activation).
6) The reactor rejects a package that fails integrity verification and does not serve its content (Clause 5, Security).
7) The reactor exposes the Monitoring HTTP API of [reactor-clause], answering GET requests under `/api/v1/introspect` for each activated package:
   a) `/api/v1/introspect/metadata` returning package metadata;
   b) `/api/v1/introspect/routes` returning served routes;
   c) `/api/v1/introspect/content-hashes` returning content hashes;
   d) `/api/v1/introspect/content-validity` returning the integrity verification outcome.
8) The reactor exposes the reactor-level introspection endpoints of [reactor-clause] — `/introspect/status`, `/introspect/config` (with secrets and URL userinfo redacted), and `/introspect/metrics` — with the specified response shapes, and the per-package endpoints `/package/<name>/status`, `/package/<name>/metadata`, and `/package/<name>/logs` (with the `lines` parameter), resolving `<name>` against the metadata names of the mounted packages and answering `404 Not Found` for unknown names.
9) When several packages are mounted, requests are dispatched by longest mount path prefix and introspection aggregates all mounted packages ([reactor-clause], Multiple mounted packages).
10) Requests to routes or endpoints with an unsupported HTTP method are answered with an appropriate HTTP error status.

A Core Reactor may additionally claim any of the module conformance classes below; their absence does not affect core conformance. A reactor that does not claim the Capsium handler-routes class shall answer requests to handler routes with `501 Not Implemented`.

Capsium signatures

An artifact or implementation conforms to the Capsium signatures class when all of the following statements are true ([module-signatures]):

1) The artifact or implementation satisfies its core class (Core Package, Core Packager, or Core Reactor).
2) The package carries a digital signature in a signature file inside the package, calculated over the canonical payload: the bytes of the files referenced by `security.integrityChecks.checksums`, concatenated in sorted key order (signatures-1, signatures-2).
3) The signature uses RSA with SHA-256 and a key of at least 2048 bits, represented by an X.509 certificate or an OpenPGP key, with the `certificateType` of `security.json` selecting the verification procedure (signatures-3, signatures-4).

4) An OpenPGP signature is an armored detached signature over the same canonical payload, with the signer's public key embedded in the package (signatures-6).
5) A packager claiming this class produces the signature and records it as specified; a reactor claiming this class verifies the signature with the public key and should reject a package whose signature does not verify (signatures-5, signatures-7).

Capsium encryption

An artifact or implementation conforms to the Capsium encryption class when all of the following statements are true ([module-encryption]):

1) The artifact or implementation satisfies its core class (Core Package, Core Packager, or Core Reactor).
2) Package encryption uses AES-256 in GCM mode, the encrypted package file carries the `.enc` extension, and `metadata.json` and `signature.json` remain unencrypted (encryption-1, encryption-2).
3) Encryption keys are distributed and stored securely, and any data encryption key (DEK) is encrypted with the recipient's public key (encryption-3, encryption-5).
4) The encryption envelope in `signature.json` declares `AES-256-GCM` with a `keyManagement` of `"RSA-OAEP-SHA256"` or `"OpenPGP"`, carrying the GCM `iv` and `authTag` and the protected DEK (`encryptedDek` or `message`), and a decryptor auto-detects the key management procedure from `keyManagement` (encryption-6).
5) Individually encrypted files are declared in the `encryption` configuration with `file`, `encryptedWith`, and `algorithm` values from the defined enumerations (encryption-4).
6) Routes serving encrypted files indicate the required decryption method, and the manifest lists the encryption details of encrypted files (encryption-7).

Capsium layered-storage

An artifact or implementation conforms to the Capsium layered-storage class when all of the following statements are true ([module-layered-storage]):

1) The artifact or implementation satisfies its core class (Core Package, Core Packager, or Core Reactor).
2) Layers are declared in `storage.json` under `storage.layers`, ordered from bottom to top, each with `path`, `writable`, and `visibility`, and the `content/` directory always constitutes the implicit bottom layer (layered-storage-1, layered-storage-2).
3) The layers are merged into a single filesystem view resolved top-first with first hit winning, missing content falls through to dependencies' exported content, and non-content paths bypass the layers (layered-storage-3, layered-storage-4, layered-storage-7).
4) Deletions are recorded as `.capsium-tombstones` JSON arrays honored at and below their layer, runtime marker files are not covered by pack-time checksums, and `private` layers are invisible to dependent packages (layered-storage-5, layered-storage-6, layered-storage-8).

Capsium composite

An artifact or implementation conforms to the Capsium composite class when all of the following statements are true ([module-composite]):

1) The artifact or implementation satisfies its core class (Core Package, Core Packager, or Core Reactor).
2) Dependencies are declared in `metadata.json` as a GUID-to-range object, and dependency resource references use the `<guid>/<path>` form with longest GUID prefix matching (composite-1, composite-2).
3) Dependency packages are resolved through the bundle → store → registry chain, selecting the newest satisfying version; a package store holds `<name>-<version>.cap` files with an optional GUID index (composite-3, composite-4).

4) Encapsulated dependencies are embedded under `packages/` with `packages/index.json`, covered by the parent checksums, re-verified at resolution, and verified against their own `security.json` at activation (composite-5).
5) On activation, dependency content becomes a lower read-only layer and only `exported` resources and routes are visible to the dependent (composite-6).
6) Route inheritance declares the dependency resource and supports remapping, response rewriting, response header enhancement, and request header supplanting with the module's semantics (composite-7).
7) Each included package and the composite package itself carry integrity hashes and should be signed (composite-8).

Capsium authentication

An artifact or implementation conforms to the Capsium authentication class when all of the following statements are true ([module-authentication]):

1) The artifact or implementation satisfies its core class (Core Package, Core Packager, or Core Reactor).
2) User credentials, password policy, account lockout, and session management follow the module's requirements, and authentication data is encrypted in transmission and storage (authentication-1, authentication-2, authentication-3).
3) `.htpasswd` files are stored securely outside the served content with securely hashed passwords (authentication-4).
4) OAuth client secrets are kept outside the package, in environment variables or server-side configuration (authentication-5).
5) A reactor claiming this class enforces `accessControl` declarations (`roles`, `authenticationRequired`) on dataset routes (authentication-6).

Capsium handler-routes

An artifact or implementation conforms to the Capsium handler-routes class when all of the following statements are true ([module-handler-routes]):

1) The artifact or implementation satisfies its core class (Core Package, Core Packager, or Core Reactor).
2) Handler routes declare `path`, `method`, and a `handler` resolving to a file in the package (handler-routes-1, handler-routes-2).
3) A reactor claiming this class executes the handler of a matching route as an ECMAScript module with a default or named `fetch` export of the shape `(request: Request) => Response | Promise<Response>`, answers unsupported methods with `405 Method Not Allowed` and an `Allow` header, and answers import or runtime failures with `502 Bad Gateway` (handler-routes-3, handler-routes-4, handler-routes-6).
4) Declared header responses are applied to handler responses, and reactors without a JavaScript execution contract answer handler routes with `501 Not Implemented` (handler-routes-5, handler-routes-7).

Capsium testing

An artifact or implementation conforms to the Capsium testing class when all of the following statements are true ([module-testing]):

1) The artifact or implementation satisfies its core class (Core Package, Core Packager, or Core Reactor).
2) Test files follow the YAML DSL: a top-level `tests` list whose entries declare `name` and `type` (testing-1).
3) Each test declares the attributes required for its type: `route` tests `url` and `expected_status`; `file` tests `path`; `data_validation` tests `format`, `data_file`, and `schema_file`; `config` tests `format` and `config_file` (testing-2 to testing-5).

4) A test runner claiming this class correctly interprets and executes the tests of all four types (testing-6).

Capsium registries

An artifact or implementation conforms to the Capsium registries class when all of the following statements are true ([module-registries]):

1) The artifact or implementation satisfies its core class (Core Package, Core Packager, or Core Reactor).
2) The registry is a directory or HTTPS base URL whose root `index.json` follows the module's schema, with package files stored relative to the registry root and their SHA-256 checksums and sizes recorded (registries-1, registries-2); network access uses HTTPS except for loopback addresses (registries-3).
3) A push validates the package, copies the `.cap` file in, recomputes the checksum and size, and rewrites `index.json` atomically (registries-4).
4) Resolution selects the newest indexed version satisfying the requested semantic version range (registries-5).
5) Installation verifies the downloaded file against the indexed checksum, rejects mismatches, and installs into a package store as `<name>-<version>.cap` with the store index updated (registries-6).
6) `capsium://` package references in mounts and dependencies resolve through the bundle → store → registry chain (registries-7).

Capsium writable-packages

An artifact or implementation conforms to the Capsium writable-packages class when all of the following statements are true ([module-writable-packages]):

1) The artifact or implementation satisfies its core class (Core Package, Core Packager, or Core Reactor).
2) A package declaring `readOnly: true` is immutable and its write requests are answered `403 Forbidden` (writable-packages-1).
3) The base package is never modified: writes go to an append-only reactor top layer, merged resolution is top-first, and deletions are recorded as `.capsium-tombstones` (writable-packages-2, writable-packages-3).
4) The dataset CRUD API behaves as specified: `201` with `Location` on creation, `200` on read and update, `204` on deletion, `400` on malformed JSON, `404` on unknown dataset or item, `405` on unsupported methods, `409` on identifier conflict, `422` with details on schema violation, and `501` for dataset kinds the reactor cannot modify; item identity follows the `id` field or the 1-based index as a string (writable-packages-4, writable-packages-5).
5) Content writes behave as specified: `PUT` creates or overwrites with on-demand routing and un-tombstoning, `DELETE` tombstones with `404` semantics (writable-packages-6).
6) The GraphQL API exposes per-dataset list queries and create/update/delete mutations, derives item types from the dataset schema or a permissive JSON scalar, and reports request failures in the `errors` array rather than as HTTP `500` (writable-packages-7).
7) Writes are visible on the next request without restart, and overlay state persists in the reactor work directory across restarts (writable-packages-8).
8) The save operation folds base and overlays into a new `<name>-<version>.cap` with the patch version incremented, regenerates `manifest.json`, `routes.json`, and `security.json`, validates the result, and returns its path and SHA-256 checksum (writable-packages-9).

Appendix A (informative) Ruby packager

The `capsium` Ruby gem (version 0.6.0, MIT licensed, copyright Ribose) is the reference packager and reactor for Capsium packages. It packages a directory into a `.cap` file, unpacks, inspects and validates packages, and includes the WEBrick-based reactor described in Annex B. This annex describes the packager side of the gem; it requires Ruby 3.2 or later.

A.1. Architecture

The gem is organized under `lib/capsium` with autoload-only loading: the top level `Capsium` module declares `autoload` for every component (`Package`, `Packager`, `Reactor`, `Cli`, `Converters`, `ThorExt`, `VERSION`), so no implementation file is loaded until the corresponding constant is first referenced.

Package domain objects are modelled with `lutaml-model` (`Lutaml::Model::Serializable`). Each configuration file of Clause 5 has a dedicated model with explicit JSON mappings:

— `metadata.json` — `Capsium::Package::MetadataData` (with a nested `Repository` model);
— `manifest.json` — the manifest model, keyed by package-relative resource path;
— `routes.json` — the routes model (`index` plus a `routes` array);
— `storage.json` — the storage model (`storage.dataSets`);
— `security.json` — the security model (`security.integrityChecks`).

The metadata model carries the field-level format rules of Clause 5 (kebab-case `name`, semantic `version`, URI `guid`, UUID `uuid`) and reports human-readable format errors; these rules drive the `validate` command.

The command-line interface is built on Thor. `Capsium::Cli` exposes the subcommands `package`, `reactor`, and `convert` — implemented as Thor subcommand classes under `lib/capsium/cli` — plus a top-level `install` command for registry installs.

Notable dependencies: `lutaml-model` (domain models), `thor` (CLI), `rubyzip` (`.cap` compression and extraction), `marcel` (MIME type detection when generating manifests), `json-schema` (dataset schema validation), `sqlite3` (SQLite datasets), `listen` and `webrick` (reactor, Annex B).

A.2. Installation

```
gem install capsium
```

Figure A.1

A.3. Usage

A.3.1. Packing a package

```
capsium package pack [--force/-f] [--bundle-deps] [--store DIR] [--registry R]
path-to-package
```

Figure A.2

`pack` generates `manifest.json` and `routes.json` when absent, writes `security.json` with a SHA-256 checksum of every package file except `security.json` itself, and compresses the directory into `{name}-{version}.cap`, with `name` and `version` taken from `metadata.json`. Without `--force`, an existing target file aborts the build with `Capsium::Packager::FileAlreadyExistsError`.

With `--bundle-deps`, the declared dependencies are resolved from the store or registry and encapsulated under `packages/` with a `packages/index.json` index ([module-composite]), producing a self-contained composite package.

EXAMPLE — Sample pack session

```
$ capsium package pack -f spec/fixtures/bare-package
Package created: bare-package-0.1.0.cap
```

A.3.2. Unpacking a package

```
capsium package unpack bare-package-0.1.0.cap [-o/--output my_bare_package]
```

Figure A.3

When `-o` is omitted, the package is unpacked into a directory named after the `.cap` file. Extraction is zip-slip safe: entries whose names would escape the destination (absolute paths, drive letters, `..` segments) are rejected with `Capsium::Packager::UnsafeEntryError`.

A.3.3. Validating a package

```
capsium package validate path-to-package-or-cap
```

Figure A.4

Validation runs a per-check report and exits with status 1 on any failure:

— metadata: required fields present and well-formed (kebab-case name, semver version, URI guid, UUID);
— manifest: every resource exists on disk;
— routes: route targets exist; dataset routes live under `/api/v1/data/`; the index resolves to an existing HTML file;
— storage: dataset sources (and schemas) exist; dataset data passes JSON-schema validation;
— security: checksums match when `security.json` is present;
— content: no external `http(s)` references in packaged content files.

A.3.4. Inspecting a package

```
capsium package info     path-to-package
capsium package metadata path-to-package
capsium package manifest path-to-package
capsium package routes   path-to-package
capsium package storage  path-to-package
```

Figure A.5

`info` prints the package path, routes, and manifest; the other commands pretty-print the corresponding configuration as JSON.

A.4. Canonical and legacy schema support

The gem reads both the canonical configuration forms of Clause 5 and the legacy (pre-0.2) gem forms, normalizing the legacy forms on read; writers emit only the canonical forms. For example, a legacy `dependencies` array of `{name, version}` objects in `metadata.json` is accepted and normalized to the canonical object form mapping each dependency GUID to its version range. The full set of accepted legacy forms and their normalization is given in annex E.

A.5. Registry operations

The gem implements the static registry of [module-registries]: publishing to a filesystem registry, and resolving and installing from a directory or an HTTPS registry (plain HTTP is accepted for loopback addresses only).

```
capsium package push example-package-1.0.0.cap --registry /srv/registry
capsium install capsium://example.com/hello-world \
    --constraint ">=1.0.0" --registry https://registry.example.com \
    --store ~/.capsium/store
```

Figure A.6

`push` validates the package, copies it into the registry, recomputes its SHA-256 checksum and size, and rewrites `index.json` atomically (temporary file plus rename). A remote registry is read-only. `install` resolves the newest version satisfying the constraint, verifies the downloaded `.cap` against the indexed checksum — rejecting it on mismatch — and installs it into the store as `<name>-<version>.cap`, updating the store index. The `--registry` and `--store` options default to the `CAPSIUM_REGISTRY` and `CAPSIUM_STORE` environment variables.

A.6. security.json generation

At pack time the gem generates `security.json` (Clause 5, Security) with `checksumAlgorithm` `SHA-256` and a `checksums` object holding the SHA-256 hex digest of every file in the package except `security.json` itself:

```
{
  "security": {
    "integrityChecks": {
      "checksumAlgorithm": "SHA-256",
      "checksums": {
        "content/index.html": "d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2",
        "metadata.json": "e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3"
      }
    }
  }
}
```

Figure A.7

When `security.json` is present, loading a package verifies it and raises `Capsium::Package::Security::IntegrityError` on mismatch.

A.7. Programmatic usage

```
require 'capsium'

package = Capsium::Package.new('example-package-1.0.0.cap')
puts package.metadata.name
errors = package.verify_integrity # typed errors, empty when valid

packager = Capsium::Packager.new
cap_file = packager.pack(package, force: true)
packager.unpack(cap_file, 'output-directory')
```

Figure A.8

The public API ships with RBS signatures in `sig/`.

Appendix B (informative) Ruby reactor

The `capsium` Ruby gem (Annex A) ships a built-in reactor, `Capsium::Reactor`, implemented on WEBrick. It activates one or more packages from local `.cap` files, package directories, or `capsium://` registry references, and serves their routes over HTTP, together with the Monitoring HTTP API of [reactor-clause].

B.1. Starting the reactor

```
capsium reactor serve my_package.cap [more.cap ...] \
    [--mount PATH=SOURCE] [--port 8864] [--store DIR] [--registry R] \
    [--workdir DIR] [--do_not_listen]
```

Figure B.1

The default port is `8864`. While serving, the reactor watches the package directory with the `listen` gem and restarts the server when files change, so edits to a package directory are picked up without manual restarts.

A source of the form `capsium://<guid>` is installed from the configured registry into the package store first and served from there ([module-registries]).

B.2. Multiple mounted packages

Several packages can be served at once, either as multiple source arguments or as repeatable `--mount PATH=SOURCE` mount maps (`{path, source}`, [reactor-clause]). When a mount path is not given, the first package is mounted at `/` and every further package at `/<metadata.name>/`; duplicate mount paths abort startup with `Capsium::Reactor::MountConflictError`. Requests are dispatched to the longest matching mount prefix, and the introspection endpoints aggregate all mounted packages.

B.3. Route serving

On startup the reactor mounts every route declared by the package's `routes.json`, plus the introspection paths, as WEBrick mount procs, and resolves each incoming request path against the package routes:

— an unknown path is answered `404 Not Found`;
— a resource route serves the referenced file with the MIME type recorded in `manifest.json` (falling back to content-based detection), and a `404` when the file is missing on disk;
— a dataset route (`{path, dataset}`) serves the dataset from `storage.json` as `application/json` under its `/api/v1/data/<id>` mount — JSON natively, YAML parsed and served as JSON — and a `404` when the dataset is unknown;
— any other route kind, including dynamic handler routes ([module-handler-routes]), is answered `501 Not Implemented`.

Unless a route declares its own headers, resource responses carry `Cache-Control: public, max-age=31536000` (one year), the reactor's configurable default; a route's `headers` replace the default for the headers they declare. Method restrictions are enforced on the introspection, data, and write APIs as specified below.

B.4. Integrity verification on load

Loading a package that contains `security.json` verifies every listed SHA-256 checksum before anything is served; a mismatch raises `Capsium::Package::Security::IntegrityError` and the package is rejected (Clause 5, Security).

B.5. Writable packages

The reactor implements the writable packages module ([module-writable-packages]) for mounts whose package is writable. Overlay state lives under the reactor work directory (`--workdir`) at `overlays/<name>/`, persists across restarts over the same work directory, and every write is visible on the next request.

Write gate	A package whose metadata declares `readOnly: true` answers every write with `403 Forbidden`.
Dataset CRUD	`POST /api/v1/data/<dataset>` creates an item (`201 Location`); `GET .../<id>`, `PUT .../<id>`, and `DELETE .../<id>` read, replace, and delete one item (`200` / `200` / `204`, `404` when absent). Malformed JSON is answered `400`, identifier conflicts `409`, schema violations `422` with the validation messages, unsupported methods `405`, and writes to SQLite datasets `501`. Items are identified by their `id` field, else by their 1-based index as a string.
Content writes	`PUT <path>` writes content into the overlay (creating the route on demand, lifting any tombstone) and `DELETE <path>` records a tombstone in `.capsium-tombstones`; the path resolves `404` thereafter, even when a lower layer holds the file.
GraphQL	A mount with at least one JSON-backed dataset exposes `POST|GET <mount>/graphql` with a list query field (optional `id:` argument) and `create<Dataset>` / `update<Dataset>` / `delete<Dataset>` mutations per dataset; item types derive from the dataset's JSON schema, falling back to a permissive `JSON` scalar. Request failures are reported in the GraphQL `errors` array.
Save	`POST /package/<name>/save` folds base package and overlays into a new `<name>-<version>.cap` with the patch version incremented: tombstones are applied, dataset logs replayed into the data files, and `manifest.json`, `routes.json`, and `security.json` regenerated. The saved package passes package validation and is unsigned; the response returns its `path`, `sha256`, `name`, and `version`.

B.6. Introspection endpoints

While serving, the reactor answers the Monitoring HTTP API ([reactor-clause]) as `application/json`. The endpoints are GET-only; other methods are answered `405 Method Not Allowed`.

```
GET /api/v1/introspect/metadata         # => {"packages": [{"name", "version",
"author", "description"}]}
GET /api/v1/introspect/routes           # => {"routes": [{"package", "routes":
[{"method", "path"}]}]}
GET /api/v1/introspect/content-hashes   # => {"contentHashes": [{"package",
"hash"}]}
```

```
GET /api/v1/introspect/content-validity # => {"contentValidity": [{"package",
"valid", "lastChecked", "signed", "encrypted", "signatureValid"?, "reason"?}]}
```

Figure B.2

`metadata` wraps the served packages' `name`, `version`, `author`, and `description` in the `packages` list shape of [reactor-clause], aggregating all mounted packages.

`routes` lists every served route with its HTTP method (defaulting to `GET`) and path.

`content-hashes` reports the SHA-256 of the `.cap` blob when the package was served from one. For a directory source there is no blob, so the hash covers the canonical (sorted-key) JSON serialization of the package content checksums — the same data `security.json` integrity checks carry.

`content-validity` re-verifies the package against `security.json` on every request and reports the outcome with a UTC `lastChecked` timestamp; `reason` lists the integrity errors when `valid` is `false`. Entries also report `signed` and `encrypted`, and `signatureValid` for signed packages.

Example `content-validity` response for a healthy package:

```
{
  "contentValidity": [
    {
      "package": "example-package",
      "valid": true,
      "lastChecked": "2024-05-28T12:34:56Z",
      "signed": true,
      "encrypted": false,
      "signatureValid": true
    }
  ]
}
```

Figure B.3

B.7. Reactor-level and per-package introspection

The reactor also answers the reactor-level and per-package introspection endpoints of [reactor-clause], GET-only and as `application/json`:

```
GET /introspect/status          # => {"status": "running", "uptime": <seconds>,
"packagesLoaded": <n>}
GET /introspect/config          # => {"port", "storeDir", "cacheControl",
"authEnabled", "registry"}
GET /introspect/metrics         # => {"uptime": <seconds>, "requestsTotal": <n>,
"requestsByStatus": {"<status>": <n>}}
GET /package/<name>/status      # => {"package", "version", "status": "loaded",
"valid"}
GET /package/<name>/metadata    # => {"name", "version", "description",
"author", "guid"}
GET /package/<name>/logs        # => {"package", "logs": [...]} (?lines=N,
default 100, clamped to 1..1000)
```

Figure B.4

`/introspect/config` redacts secrets: userinfo in configured URLs is stripped and `deploy.json` secrets never appear. The per-package endpoints resolve `<name>` against the metadata names of all mounted packages and answer `404 Not Found` for unknown names.

Appendix C
(informative)
Apache httpd reactor

No Apache httpd implementation of a Capsium reactor exists at the time of writing. This annex provides guidance for realizing one, mapping the reactor responsibilities of [reactor-clause] onto standard Apache httpd modules. Because a Capsium package is static by design (Clause 4), most of the serving behavior can be expressed in declarative httpd configuration; only integrity verification and the Monitoring HTTP API require external tooling or a dynamic backend.

C.1. Extracting the package to a docroot

A `.cap` file is a ZIP archive (Clause 5, Packaging options). Deployment starts by extracting it into a directory that becomes (part of) the Apache document root:

```
mkdir -p /var/www/capsium/example-package-1.0.0
cd /var/www/capsium/example-package-1.0.0
unzip /srv/capsium/packages/example-package-1.0.0.cap
```

Figure C.1

Extraction should reject archive entries whose paths would escape the destination directory (absolute paths, `..` segments), as required of packagers in [management-clause]. After extraction, the served content tree is available at the docroot and the configuration files (`metadata.json`, `manifest.json`, `routes.json`, `security.json`) are available for deriving the site configuration.

C.2. Integrity pre-verification at deploy time

Apache httpd itself does not verify `security.json`. Integrity verification (Clause 5, Security) should therefore be performed at deploy time by external tooling, before the extracted directory is published: recalculate the SHA-256 of every extracted file and compare it against `security.integrityChecks.checksums`; refuse to publish the docroot on any mismatch. The `capsium package validate` command (Annex A) performs exactly this check, as does a short script in any language with a SHA-256 library.

C.3. Mapping routes.json to Alias and mod_rewrite

Each resource route in `routes.json` maps a URL path to a package-relative file. Static mappings translate directly into `Alias` directives or, for rule shapes such as the dual HTML routes of Clause 5 (base name and full file name), into `mod_rewrite` rules:

```
Alias "/styles.css" "/var/www/capsium/example-package-1.0.0/content/styles.css"

RewriteEngine On
# Dual HTML routes: /page serves content/page.html
RewriteCond "%{DOCUMENT_ROOT}/content/$1.html" -f
RewriteRule "^/([^/]+)$" "/content/$1.html" [PT]
```

Figure C.2

The index route of `routes.json` maps to `DirectoryIndex`. Dataset routes under `/api/v1/data/` (Clause 5, Dataset routes) map to the JSON rendering of the dataset: for file-backed datasets this can again be an `Alias` to a pre-generated JSON file; database-backed datasets require a small dynamic backend (see the introspection paragraph below).

C.4. Serving manifest-derived MIME types

`manifest.json` records the MIME type of every resource (Clause 5, Manifest file). httpd's own `mime.types` covers the common types; package-specific or unusual types recorded in the manifest can be added with `mod_mime`:

```
AddType application/vnd.capsium.package .cap
AddType text/javascript .js
```

Figure C.3

C.5. Cache-Control via mod_headers

Reactors conforming to [reactor-clause] serve static resources with a long-lived `Cache-Control` policy by default. With `mod_headers`:

```
<Directory "/var/www/capsium/example-package-1.0.0/content">
  Header set Cache-Control "public, max-age=31536000"
</Directory>
```

Figure C.4

Per-route header declarations in `routes.json` (Clause 5, Header responses) translate into more specific `Header` directives scoped by `<Location>`.

C.6. Basic authentication

The Apache `passwd` authentication defined for packages ([module-authentication]) is native httpd functionality. A package that declares `.htpasswd` basic authentication maps to:

```
<Directory "/var/www/capsium/example-package-1.0.0/content">
  AuthType Basic
  AuthName "Restricted Access"
  AuthUserFile /etc/httpd/capsium/example-package.htpasswd
  Require valid-user
</Directory>
```

Figure C.5

The `.htpasswd` file is managed with the `htpasswd` utility, must be stored outside the served docroot, and passwords should be hashed with bcrypt ([module-authentication]). OAuth authentication defined by a package requires a dynamic backend or a module such as `mod_auth_openidc`, with the client secret kept outside the package as specified in [module-authentication].

C.7. Exposing the introspection API

The Monitoring HTTP API ([reactor-clause]) is dynamic and cannot be served from static files alone, because `content-validity` re-verifies the package at request time. Options for exposing it behind httpd include:

— a CGI or FastCGI script that reads the extracted package's configuration files and recomputes the verification on demand;
— a small application server (for example the Ruby reactor of Annex B running alongside httpd) reverse-proxied with `mod_proxy`:

```
ProxyPass        "/api/v1/introspect" "http://127.0.0.1:8864/api/v1/introspect"
```

```
ProxyPassReverse "/api/v1/introspect" "http://127.0.0.1:8864/api/v1/introspect"
```

Figure C.6

Whichever backend is chosen, the four endpoints of [reactor-clause] — `/api/v1/introspect/metadata`, `/routes`, `/content-hashes`, and `/content-validity` — should answer GET requests with the JSON shapes defined there, and reject other methods with `405`.

Appendix D
(informative)
nginx (OpenResty) reactor

The `capsium-lua` project (version 0.4.0, MIT licensed) is a Capsium reactor built on OpenResty, the Lua-enabled nginx distribution. It serves Capsium packages (`.cap` files) directly through nginx with Lua: a `.cap` file dropped into the packages directory is extracted on demand, verified against `security.json`, and routed, with optional per-mount configuration for paths, domains, headers, caching, and CORS. Mounts may also reference packages by `capsium://` URI, pulling them from a static registry ([module-registries]). Serving is read-only: the reactor has no write API.

D.1. Architecture

The implementation keeps a single framework-agnostic core and a thin nginx glue layer, with no duplicated logic:

```
lib/capsium/                  Framework-agnostic core (the rock)
├── utils.lua                 Pure helpers (tables, JSON files, paths,
│                             URL userinfo redaction)
├── mime.lua                  MIME type detection (text/javascript per RFC 9239)
├── csv.lua                   Minimal CSV parser (dataset support)
├── yaml.lua                  Minimal YAML parser (dataset support)
├── semver.lua                Semantic versioning ranges
├── crypto.lua                Decryption support (encrypted packages)
├── log_buffer.lua            Per-worker ring buffer for request logs
├── reactor.lua               Reactor core: package management + introspection
├── registry.lua              Static registry client (resolve/install)
├── auth/                     Authentication (basic auth, OAuth2)
├── package/
│   ├── package.lua           Package model (OOP): load/normalize/resolve
│   ├── extractor.lua         Atomic .cap extraction (tmp dir + rename)
│   ├── router.lua            Schema normalization + route auto-generation
│   ├── security.lua          security.json parsing + SHA-256 verification
│   ├── store.lua             Package store (dependency resolution)
│   ├── composite.lua         Composite packages (dependency content)
│   └── decrypter.lua         Package decryption
└── adapters/
    ├── nginx.lua             fs/zip adapters (LuaFileSystem + lua-zip)
    └── hash.lua              SHA-256 (resty.sha256 in OpenResty,
                                pure-Lua fallback elsewhere)

lua/capsium/                  nginx glue (only code that touches ngx)
├── init.lua                  Entry point: wires adapters into the core,
│                             request handling, introspection API
└── config.lua                config.json loading + mount resolution
```

Figure D.1

The core never references ngx; the glue never implements domain logic. Other platforms can plug in their own filesystem, zip, and hash adapters.

D.2. Configuration

Configuration is read from `$CAPSIUM_CONFIG_PATH`, else the first existing of `/etc/capsium/config.json`, `/etc/capsium/nginx/config.json`, `/var/lib/capsium/config.json`, `./config.json`. Top-level keys:

`package_dir`	Directory containing `.cap` files (default `/var/lib/capsium/packages`).
`extract_dir`	Extraction target (default `/var/lib/capsium/extracted`).
`cache_enabled`	Use the `capsium_cache` shared dict to cache introspection responses (default `true`).
`cache_ttl`	TTL in seconds for the shared-dict cache (default `3600`).
`packages_config_dir`	Optional directory with per-package `<name>.json` mount configs, merged with `mounts` at startup.
`registry`	Default static registry for `capsium://` mount sources ([module-registries]); also settable with the `CAPSIUM_REGISTRY` environment variable.
`store_dir`	Package store into which registry-resolved packages are installed; also settable with the `CAPSIUM_STORE` environment variable.
`mounts`	Array of mount entries (below).

Example mount entries:

```
{
  "package_dir": "/var/lib/capsium/packages",
  "extract_dir": "/var/lib/capsium/extracted",
  "cache_enabled": true,
  "cache_ttl": 3600,
  "registry": "/var/lib/capsium/registry",
  "mounts": [
    {
      "package": "mn-samples-iso-0.1.0.cap",
      "path": "/app",
      "domain": "example.com",
      "options": {
        "cache_ttl": 7200,
        "headers": {
          "X-Frame-Options": "SAMEORIGIN",
          "X-Content-Type-Options": "nosniff"
        }
      }
    },
    {
      "package": "api-1.0.0.cap",
      "path": "/api",
      "options": {
        "cors": {
          "allowed_origins": ["https://example.com"],
          "allowed_methods": ["GET"],
          "allowed_headers": ["Content-Type"],
          "max_age": 600
        }
      }
    },
    {
      "package": "capsium://example.com/registry-app",
      "path": "/registry-app",
      "version": ">=1.0.0"
    }
  ]
```



```
}
```

Figure D.2

Mount entry keys:

`package`	Package filename (`.cap` extension optional), or a `capsium://<guid>` registry reference (below).
`path`	Mount path prefix (default `/capsium/<package-name>`).
`domain`	Virtual host. Advisory: the mount answers on all hosts, and is preferred for the named host when mounts overlap.
`version`	Semantic version range for a `capsium://` source (default *).
`registry`	Per-mount registry override for a `capsium://` source.
`store`	Per-mount store override for a `capsium://` source.
`options.cache_ttl`	Static `Cache-Control` max-age override for this mount (default `31536000`, giving `public, max-age=31536000`).
`options.headers`	Extra response headers applied to every response from this mount; an explicit `Cache-Control` here wins over `cache_ttl`.
`options.cors`	CORS policy: `allowed_origins` (array, * supported), `allowed_methods`, `allowed_headers`, `expose_headers`, `max_age`. Preflight `OPTIONS` requests are answered with `204` when CORS is configured.
`options.encryption.privateKeyPath`	Private key used to serve an encrypted package ([module-encryption]).

Every package in `package_dir` is always also available at `/capsium/<package-name>` without configuration; `/api/v1/introspect`, `/introspect`, and `/package` are reserved. Mounts are matched longest-path-first, with the `domain` preference applied first.

D.3. Registry pull

A mount whose `package` is a `capsium://<guid>` URI is resolved through the static registry ([module-registries]): on the first request the reactor resolves the newest indexed version satisfying the mount's `version` constraint, downloads the `.cap` from the registry — HTTPS only, except for loopback addresses, with the scheme re-validated on every redirect — and verifies its SHA-256 against the index entry, rejecting the download on mismatch. The verified file is installed atomically (write-then-rename) into the store as `<name>-<version>.cap` and served from there; a store file whose checksum already matches the index is reused without download. Composite package dependencies are resolved from the store ([module-composite]).

D.4. Extraction and integrity verification

Packages are extracted lazily, on first request. Extraction is atomic: the archive unpacks into a temporary directory that is renamed into place only after `metadata.json` parses and, when `security.json` is present, every listed file's SHA-256 verifies (Clause 5, Security). A package that fails verification is rejected with a 5xx response carrying the reason, and nothing is left at the final extraction path. Archive entries containing `..` path segments are refused (zip-slip protection). Re-extraction happens automatically when the `.cap` file is newer than the extracted tree.

D.5. Manifest-driven routing

Both the canonical configuration forms of Clause 5 and the legacy gem forms are accepted and normalized on read. When `routes.json` is absent, routes are auto-generated from `manifest.json` per the routing rules of Clause 5:

— `index` defaults to `content/index.html`, also routed at `/`;
— every manifest resource under `content/` is routed relative to `content/`;
— HTML files get dual routes, `/page` and `/page.html`;
— every dataset gets `/api/v1/data/<id>`.

Package routes are GET-only: other methods receive `405` with `Allow: GET, HEAD`; HEAD is served like GET without a body. Dynamic `handler` routes ([module-handler-routes]) are accepted by the parser but answered with `501 Not Implemented`. Responses carry the media type derived from the file name extension when known — `.js` is `text/javascript` per RFC 9239 — with the `manifest.json` entry as the fallback. Dataset sources are served as `application/json` — `.json` served as-is, `.csv` converted to JSON objects, `.yaml` parsed by the bundled YAML parser and served as JSON. SQLite datasets are not supported by this reactor; their routes are answered with an error response naming the unsupported dataset.

D.6. Introspection API

The reactor answers the Monitoring HTTP API ([reactor-clause]), GET only:

`/api/v1/introspect/metadata`	`{packages: [{name, version, author, description}]}`
`/api/v1/introspect/routes`	`{routes: [{package, routes: [{method, path}]}]}`
`/api/v1/introspect/content-hashes`	`{contentHashes: [{package, hash}]}` — SHA-256 of the `.cap` blob.
`/api/v1/introspect/content-validity`	`{contentValidity: [{package, valid, lastChecked, reason?, signed?, encrypted, signatureValid?}]}` — the actual integrity verification result, with the signature and encryption posture of the package.

All introspection endpoints lazily extract and load packages on demand, so a freshly started reactor reports every `.cap` in `package_dir` (cold-start introspection). Responses are cached in the `capsium_cache` shared dict with the configured `cache_ttl`. Of these endpoints, only `/api/v1/introspect` is guarded when a package mounted at `/` declares authentication ([module-authentication]).

D.7. Reactor-level and per-package introspection

The reactor also answers the reactor-level and per-package endpoints of [reactor-clause], GET only:

`/introspect/status`	`{status: "running", uptime, packagesLoaded}` — `packagesLoaded` counts the configured mounts.
`/introspect/config`	The effective configuration, built from a whitelist: `authentication` secrets and `encryption.privateKeyPath` can never appear, and userinfo in the registry URL is redacted.

`/introspect/metrics`	`{uptime, requestsTotal, requestsByStatus}` — per status code counts from the `capsium_metrics` shared dict.
`/package/<name>/status`	`{package, version, status: "loaded", valid}`.
`/package/<name>/metadata`	`{name, version, description, author, guid}`.
`/package/<name>/logs`	`{package, logs: [...]}` from a per-worker ring buffer; `?lines=N` selects the number of lines (default 100, clamped to 1..1000).

The per-package endpoints resolve `<name>` against the metadata names of all mounted packages — resolving registry mounts lazily first — and answer `404` for unknown names.

D.8. Docker deployment

The project ships a Dockerfile whose base image is pinned: `openresty/openresty:1.27.1.2-12-alpine-fat` (the `BASE_IMAGE` build arg). A typical deployment mounts the packages and configuration directories into the container:

```
docker build -t capsium-nginx .
docker run -d -p 8080:80 \
  -v $PWD/packages:/var/lib/capsium/packages \
  -v $PWD/config:/etc/capsium \
  capsium-nginx
```

Figure D.3

Appendix E
(informative)
Legacy configuration forms

Earlier revisions of this document and early implementations used configuration forms that predate the canonical schemas of Clause 5. This annex collects those legacy forms and defines how a reader normalizes them to the canonical schemas.

A conformant reader shall accept every legacy form in this annex and normalize it on read as specified. A conformant writer shall emit only the canonical forms of Clause 5: the legacy forms exist for backward compatibility on read and shall not be produced.

E.1. Legacy manifest

Legacy form `manifest.json` with a `content` array of `file` / `mime` entries:

```
{
  "content": [
    { "file": "content/index.html", "mime": "text/html" },
    { "file": "content/styles.css", "mime": "text/css" }
  ]
}
```

Figure E.1

Normalization Each entry becomes a resource of the canonical manifest (Clause 5, Manifest file), keyed by its `file` path, with `mime` mapped to `type` and `visibility` defaulting to `exported`:

```
{
  "resources": {
    "content/index.html": { "type": "text/html", "visibility": "exported" },
    "content/styles.css": { "type": "text/css", "visibility": "exported" }
  }
}
```

Figure E.2

E.2. Legacy routes

Legacy form 1 `routes.json` with route entries whose target is a nested `target` object carrying `file` or `dataset`:

```
{
  "routes": [
    { "path": "/", "target": { "file": "content/index.html" } },
    { "path": "/api/v1/data/animals", "target": { "dataset": "animals" } }
  ]
}
```

Figure E.3

Normalization The `target` object is folded into the route entry: `file` becomes `resource`, `dataset` becomes `dataset` (Clause 5, Resource routing):

```
{
  "routes": [
    { "path": "/", "resource": "content/index.html" },
    { "path": "/api/v1/data/animals", "dataset": "animals" }
```



```
  ]
}
```

Figure E.4

Legacy form 2 — `routes.json` with `routes` as an object keyed by path, each value describing the target and options of the route:

```
{
  "routes": {
    "/": { "resource": "content/index.html" },
    "/api/v1/data/animals": { "dataset": "animals" }
  }
}
```

Figure E.5

Normalization — The object is converted to the canonical array form: each key becomes the path of a route entry and the value its remaining attributes.

E.3. Legacy storage

Legacy form — `storage.json` with a `datasets` array of `name`, `source`, `format`, and `schema` entries:

```
{
  "datasets": [
    {
      "name": "animals",
      "source": "data/animals.json",
      "format": "json",
      "schema": "data/animals.schema.json"
    }
  ]
}
```

Figure E.6

Normalization — Each entry becomes a dataset of the canonical `storage.dataSets` object (Clause 5, Storage), keyed by `name`. `source` is kept; a present `schema` becomes `schemaFile` with `schemaType` `json-schema`; `format` is dropped — the dataset format is derived from the source file extension:

```
{
  "storage": {
    "dataSets": {
      "animals": {
        "source": "data/animals.json",
        "schemaFile": "data/animals.schema.json",
        "schemaType": "json-schema"
      }
    }
  }
}
```

Figure E.7

E.4. Legacy dependencies

Legacy form `metadata.json` with dependencies as an array of `name` / `version` objects:

```
{
  "dependencies": [
    { "name": "capsium://example.com/other-pkg", "version": ">=1.0.0" }
  ]
}
```

Figure E.8

Normalization The array is converted to the canonical object form (Clause 5.3.4): each entry's `name` becomes a GUID key and its `version` the version range:

```
{
  "dependencies": {
    "capsium://example.com/other-pkg": ">=1.0.0"
  }
}
```

Figure E.9

Bibliography

[1] IETF RFC 6749, Internet Engineering Task Force (committee). *The OAuth 2.0 Authorization Framework*. 2012. RFC Publisher. https://www.rfc-editor.org/info/rfc6749.

[2] IETF RFC 7636, BRADLEY, J., N. AGARWAL and Internet Engineering Task Force. *Proof Key for Code Exchange by OAuth Public Clients*. 2015. RFC Publisher. https://www.rfc-editor.org/info/rfc7636.

[3] IETF RFC 4648, JOSEFSSON, S. *The Base16, Base32, and Base64 Data Encodings*. 2006. RFC Publisher. https://www.rfc-editor.org/info/rfc4648.

[4] IETF RFC 6838, FREED, N., J. KLENSIN, T. HANSEN and Internet Engineering Task Force. *Media Type Specifications and Registration Procedures*. 2013. RFC Publisher. https://www.rfc-editor.org/info/rfc6838.

[5] IETF RFC 8259, Internet Engineering Task Force (committee). *The JavaScript Object Notation (JSON) Data Interchange Format*. 2017. RFC Publisher. https://www.rfc-editor.org/info/rfc8259.

[6] IETF RFC 3986, BERNERS-LEE, T., R. FIELDING and L. MASINTER. *Uniform Resource Identifier (URI): Generic Syntax*. 2005. RFC Publisher. https://www.rfc-editor.org/info/rfc3986.

[7] JSON Schema 2020-12, *JSON Schema: A Media Type for Describing JSON Documents*. JSON Schema Working Group. https://json-schema.org/draft/2020-12/json-schema-core

[8] Apache htpasswd, *htpasswd — Manage user files for basic authentication*. The Apache Software Foundation. https://httpd.apache.org/docs/current/programs/htpasswd.html

[9] OCI image-spec, *Open Container Initiative Image Format Specification*. Open Container Initiative. https://github.com/opencontainers/image-spec

[10] W3C Web Bundles, *Web Bundles* (W3C draft). W3C Web Packaging Working Group. https://w3c.github.io/webpackage/draft-yasskin-wpack-bundled-exchanges.html

Common architecture for portable secure information interchange and unified management (Capsium)

1. Scope

This document describes Capsium, a modular framework for the secure and efficient interchange of multi-format information within interoperable and portable packages.

This document specifies requirements of the Capsium framework and its components:

— Capsium packages: standardized unit of information interchange in the Capsium framework.
— Capsium reactors: server-side or user-side software that allows deployment of a Capsium package.
— Capsium HTTP API: user-facing HTTP API implemented by a Capsium reactor.

This document also provides:

— Guidelines for the utilization of the Capsium framework.
— Examples for implementing components of the Capsium framework.

2. Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 21320-1:2015, International Organization for Standardization (committee). *Information technology — Document Container File — Part 1: Core*. First edition. 2015. Geneva: International Organization for Standardization and International Electrotechnical Commission. https://www.iso.org/standard/60101.html.

NIST FIPS 180-4 fpd, National Institute of Standards and Technology. *Secure Hash Standard (SHS)*. Edition Revision 4. 2012. Gaithersburg. https://csrc.nist.gov/pubs/fips/180-4/final.

NIST SP 800-38D fpd, DWORKIN, Morris. *Recommendation for Block Cipher Modes of Operation — Galois/Counter Mode (GCM) and GMAC*. 2007. Gaithersburg. https://csrc.nist.gov/pubs/sp/800/38/d/final.

IETF RFC 8017, KALISKI, B., J. JONSSON and A. RUSCH. *PKCS #1: RSA Cryptography Specifications Version 2.2*. 2016. RFC Publisher. https://www.rfc-editor.org/info/rfc8017.

IETF RFC 9110, Internet Engineering Task Force (committee). *HTTP Semantics*. 2022. RFC Publisher. https://www.rfc-editor.org/info/rfc9110.

SemVer 2.0.0, *Semantic Versioning 2.0.0*. https://semver.org/spec/v2.0.0.html

SPDX License List, *SPDX License List*. Linux Foundation. https://spdx.org/licenses/

3. Terms and definitions

For the purposes of this document, the following terms and definitions apply.

3.1.
Capsium package
A structured bundle of multi-format data and resources that can be securely and efficiently interchanged and deployed across different platforms. Capsium packages are designed to be portable and interoperable, ensuring seamless data transfer and application deployment.

3.2.
Capsium reactor
A component responsible for managing and deploying Capsium packages. The Capsium reactor can be implemented directly by a web browser or installed on user machines or servers. It eliminates the need for a traditional web server by mimicking its file-serving capabilities.

3.3.
Capsium HTTP API
3.4.
Non-application website
A website that does not require server-side logic or dynamic content generation. These websites consist primarily of static resources such as HTML, CSS, and JavaScript files and are deployed using the Capsium framework without the need for a traditional web server.

3.5.
Single-page application (SPA)
A web application that loads a single HTML page and dynamically updates the content as the user interacts with the app. SPAs provide a more fluid and responsive user experience by reducing server dependency. Capsium supports the deployment and optimization of SPAs.

3.6.
Data immutability
The property of data that ensures it remains unchanged and secure from tampering after it has been created. Capsium guarantees data immutability through the use of advanced cryptographic techniques, ensuring the integrity and trustworthiness of the data.

3.7.
Interoperability
The ability of different systems, platforms, and applications to communicate and work together effectively. Capsium supports a wide range of formats and protocols, enabling seamless data exchange and integration across diverse environments.

3.8.
Portability
The capability of a system or application to be easily transferred and used across different environments without compromising security or functionality. Capsium packages are designed to be portable, facilitating easy migration and deployment.

3.9.
Compliance and auditing
The adherence to regulatory requirements and the ability to track and monitor data access and modifications. Capsium includes features that ensure compliance with relevant standards and provide robust auditing capabilities to maintain data integrity and security.

3.10.
Encryption
The process of converting data into a coded format to prevent unauthorized access. Capsium employs advanced encryption techniques to ensure that the data within its packages is securely stored and transferred, protecting sensitive information from breaches.

3.11.
Key management
The administration of cryptographic keys, which includes their generation, exchange, storage, use, and replacement. In the context of Capsium, key management is crucial for maintaining the security of encrypted data and ensuring that only authorized entities can access or modify the data.

3.12.
Static website
A website consisting of fixed content that does not change unless manually updated. Static websites are composed of HTML, CSS, and JavaScript files and do not require server-side processing. Capsium aids in the deployment of static websites by packaging all necessary resources into a single, portable bundle.

3.13.
Virtualization
The creation of a virtual version of something, such as an operating system, a server, a storage device, or network resources. Capsium packages reduce the need for virtualization by allowing applications to run directly in the browser or on the client machine, simplifying deployment and reducing resource requirements.

3.14.
Package management
The process of handling software packages, including their installation, upgrade, configuration, and removal. Capsium's package management capabilities ensure that Capsium packages can be efficiently deployed and maintained, streamlining the application lifecycle.

3.15.
Browser-implemented reactor
A Capsium reactor that is directly implemented by the web browser, enabling it to serve Capsium packages without the need for additional server infrastructure. This approach leverages the browser's native capabilities to handle package deployment and management.

3.16.
Serverless architecture
A design pattern where the server management and infrastructure concerns are abstracted away from the developer. Capsium supports serverless architectures by allowing applications to be deployed and run without a traditional server, relying instead on the Capsium reactor.

3.17.
Cryptographic techniques
Methods used to secure information and communications through the use of codes, ensuring that only those for whom the information is intended can read and process it. Capsium utilizes cryptographic techniques to maintain data security and integrity within its packages.

3.18.
Interchange format
A standardized format used for exchanging data between different systems or platforms. Capsium defines a specific interchange format to ensure that its packages can be seamlessly transferred and utilized across various environments.

3.19.
Multi-format information
Data that exists in various formats, such as text, images, video, and structured data. Capsium packages are designed to handle multi-format information, ensuring that diverse types of data can be securely and efficiently bundled together and deployed.

3.20.
Deployment
The process of distributing and installing software or data packages in a specific environment. Capsium streamlines deployment by allowing packages to be easily transferred and installed without the need for extensive configuration or server infrastructure.

3.21.
Regulatory compliance
Adhering to laws, regulations, and guidelines relevant to the handling and protection of data. Capsium helps organizations maintain regulatory compliance by providing tools and features that ensure data security, integrity, and traceability.

3.22.
File-serving capabilities
The ability of a server or system to deliver files to clients upon request. In the context of Capsium, the reactor mimics traditional file-serving capabilities, allowing it to serve packaged resources directly to the browser or client machine.

3.23.
Resource bundling
The process of combining multiple files and resources into a single package. Capsium facilitates resource bundling, enabling efficient transfer and deployment of all necessary components of a web application or website.

3.24.
Package integrity
The assurance that a package has not been altered or tampered with since its creation. Capsium ensures package integrity through cryptographic signatures and other security measures, guaranteeing that the contents of a package remain unchanged during transfer and deployment.

3.25.
Transferability
The ease with which data or applications can be moved from one environment to another. Capsium enhances transferability by providing a standardized package format that can be easily migrated across different platforms and systems.

3.26.
Static content
Web content that does not change and is delivered to the user exactly as stored. Capsium supports the deployment of static content by packaging it into a portable format that can be served without the need for dynamic processing.

3.27.
Advanced cryptography
The use of sophisticated encryption algorithms and techniques to protect data. Capsium employs advanced cryptography to ensure that the data within its packages is secure from unauthorized access and tampering.

3.28.
Data migration
The process of moving data from one system or environment to another. Capsium simplifies data migration by providing a portable package format that facilitates the transfer of data and resources across different platforms.

3.29.
Scalable deployment
The ability to efficiently deploy applications and data across a varying number of environments and users. Capsium supports scalable deployment by providing a flexible package format and reactor that can handle deployments of any size.

3.30.
Platform independence
The ability of software or data to operate on various hardware and operating systems without requiring modification. Capsium ensures platform independence by using standardized formats and protocols, allowing its packages to be used across different environments seamlessly.

3.31.
Dependency management
The process of handling and resolving the dependencies required by software applications or packages. Capsium includes mechanisms for managing dependencies within its packages, ensuring that all necessary components are available and properly configured during deployment.

3.32.
Immutable data
Data that cannot be altered once it has been created. Capsium guarantees immutability through cryptographic methods, making sure that data within a package remains unchanged and secure from tampering.

3.33.
Data integrity
The accuracy and consistency of data over its lifecycle. Capsium ensures data integrity by using cryptographic techniques to protect data from unauthorized alterations, ensuring reliable and trustworthy information.

3.34.
Secure interchange
The safe and protected exchange of data between different systems or platforms. Capsium facilitates secure interchange by using advanced encryption and ensuring that packages are transferred without compromising their integrity or confidentiality.

3.35.
Application lifecycle
The entire process of developing, deploying, maintaining, and eventually decommissioning an application. Capsium supports the application lifecycle by providing tools and features that streamline deployment, maintenance, and updates of packaged applications.

3.36.
Server dependency
The reliance on a server to provide resources, process requests, and manage data. Capsium reduces server dependency by enabling applications and websites to be deployed and run using the Capsium reactor, which can function without a traditional server.

3.37.
Resource optimization
The process of improving the efficiency and performance of resources used by an application or system. Capsium supports resource optimization by packaging resources in a way that reduces load times and minimizes server demands.

3.38.
Data traceability
The ability to track the history, usage, and location of data over its lifecycle. Capsium includes features that enhance data traceability, ensuring that data access and modifications can be monitored and audited for compliance and security purposes.

3.39.
Data security
The protection of data from unauthorized access, corruption, or theft. Capsium ensures data security through encryption, key management, and other protective measures, maintaining the confidentiality and integrity of packaged data.

3.40.
Audit trail
A record of all actions and changes made to data, providing transparency and accountability. Capsium supports the creation of audit trails, helping organizations monitor data access and modifications for compliance and security.

3.41.
Browser-native deployment
The ability to deploy applications and resources directly within a web browser without requiring additional plugins or software. Capsium supports browser-native deployment, leveraging the browser's capabilities to handle and serve packaged data and applications.

3.42.
Dynamic content
Web content that changes based on user interactions or other conditions. While Capsium primarily targets static and SPA content, it can support dynamic content through appropriate integration with client-side scripting.

3.43.
Lightweight deployment
A deployment method that minimizes resource usage and overhead, making it suitable for environments with limited resources. Capsium supports lightweight deployment by packaging applications and data in an efficient, compact format.

3.44.
Multi-platform support
The ability to operate across various operating systems, devices, and environments. Capsium ensures multi-platform support by adhering to standardized formats and protocols, allowing its packages to function seamlessly across different systems.

3.45.
Portable package
A self-contained bundle that includes all necessary resources and data, designed to be easily transferred and deployed across different environments. Capsium packages are inherently portable, facilitating straightforward migration and deployment.

3.46.
Client-side processing
The execution of operations on the user's device rather than on a server. Capsium supports client-side processing by enabling web applications to run directly in the browser, reducing the need for server interactions.

3.47.
Cross-platform compatibility
The ability of software or data to work on various operating systems and devices without requiring modifications. Capsium ensures cross-platform compatibility by using standardized formats and protocols, making its packages usable across different platforms.

3.48.
Version control
A system for managing changes to documents, programs, and other information stored as computer files. Capsium integrates version control mechanisms to help track changes, manage different versions, and ensure consistency of packaged data.

3.49.
Secure deployment
The practice of deploying applications and data in a manner that ensures their security throughout the process. Capsium supports secure deployment by using encryption and other security measures to protect packages from tampering and unauthorized access.

3.50.
Configuration management
The process of handling changes in software, hardware, documentation, and other components. Capsium includes features for configuration management, ensuring that packages are correctly configured and maintained throughout their lifecycle.

3.51.
Integrity check
A method to verify that data has not been altered or tampered with. Capsium performs integrity checks using cryptographic signatures, ensuring the authenticity and consistency of the data within its packages.

3.52.
Modular framework
A design approach that divides a system into smaller parts, or modules, that can be independently created and then used in different systems. Capsium is a modular framework, allowing components to be added, removed, or updated without affecting the whole system.

3.53.
User authentication
The process of verifying the identity of a user. Capsium supports user authentication to ensure that only authorized individuals can access and interact with the contents of a package.

3.54.
Data encapsulation
The bundling of data with the methods that operate on that data, restricting direct access to some of the object's components. Capsium utilizes data encapsulation to protect the integrity and security of the packaged data.

3.55.
Environment abstraction
The separation of application logic from the underlying hardware and software environment. Capsium provides environment abstraction by allowing packages to operate independently of the specific details of the deployment environment.

3.56.
Resilience
The ability of a system or application to recover quickly from failures and continue to function. Capsium enhances resilience by packaging applications in a way that minimizes dependencies and facilitates recovery and redeployment.

3.57.
Scalability
The capacity to handle increasing amounts of work or to be readily enlarged. Capsium supports scalability by ensuring that its packages can be deployed and managed efficiently, regardless of the scale of the deployment.

3.58.
Trusted execution
The assurance that code and data are executed in a secure environment, protected from unauthorized access and tampering. Capsium supports trusted execution through the use of secure packaging and deployment mechanisms.

3.59.
Content delivery
The process of distributing digital content to users. Capsium optimizes content delivery by bundling resources into efficient packages that can be served directly by the browser or client machine.

3.60.
Content encapsulation
The practice of bundling content with the necessary metadata and resources to ensure it can be used independently of its original environment. Capsium uses content encapsulation to create portable packages that can be deployed and used across different systems.

3.61.
Application sandboxing
The technique of running applications in a restricted environment to limit their access to system resources and data. Capsium can support application sandboxing by enabling packages to run in isolated environments, enhancing security and control.

3.62.
Metadata management
The process of handling metadata, which is data that describes other data. Capsium includes features for metadata management, ensuring that the necessary information about packaged data and resources is available and properly maintained.

3.63.
Data lifecycle management
The process of managing data from its creation to its eventual disposal. Capsium includes features for data lifecycle management, ensuring that data within its packages is properly handled, maintained, and disposed of according to best practices and regulatory requirements.

4. Capsium framework

The Capsium framework provides a comprehensive set of principles, features, and use cases that define its architecture and functionality. This clause outlines the essential components and concepts that make up the framework, ensuring a clear understanding of its capabilities and applications.

4.1. General

Capsium (Common architecture for portable secure information interchange and unified management) is an innovative technology framework designed to facilitate the interoperable and portable deployment of lightweight, web-compatible, interactive data packages. The framework supports the packaging and deployment of static websites, which can be hosted by any cloud file hosting service with minimal web serving functionality, such as AWS S3 or GitHub Pages.

The core concept of Capsium is to enable the packaging of static websites into deployable objects, called Capsium packages. These packages are self-contained, size-efficient, and secure, providing all necessary resources and metadata to be served by a simple web server.

4.2. Principles of the framework

The Capsium framework is based on several key principles that ensure its effectiveness and reliability:

Ease of use	The deployable object can be easily built, inspected, extracted, and deployed.
Cross-platform deployment	The deployable object can be deployed across multiple platforms without modification.
Static nature	The deployable object is static, meaning it does not require server-side processing to function.

Size efficiency	The deployable object is designed to be as small and efficient as possible.
Integrity	The deployable object cannot be corrupted, ensuring data integrity.
Self-containment	The deployable object is self-contained, including all necessary routes and redirects for the static site.
Versioning	The deployable object can be versioned, allowing for efficient updates and rollbacks.
Dependencies	The deployable object can require other deployable objects, enabling modularity and extensibility.
File system support	The deployable object has a file system that supports all types of files and provides immutable, layered versioning.
Deployment API compatibility	The deployment API can be easily implemented by common web servers such as Apache and nginx.
Compliance	The deployment API complies with common expectations and supports fetching of metadata and other introspection features.

4.3. Key features of the framework

The Capsium framework includes several key features that enhance its functionality and usability:

Capsium package	A deployable object that is a compressed, single-file package containing all files necessary for a static site.
Capsium filesystem	A file system within the Capsium package, representing the file/folder hierarchy as it will be served by the package's external API.
Capsium reactor	A Capsium-enabled web server that activates and deploys the Capsium package.
Activation	The process by which a reactor loads the content of a Capsium package and serves its routes to a web address.

These features enable the efficient creation, deployment, and management of web-compatible, interactive data packages.

4.4. Use cases of the framework

The Capsium framework supports a variety of use cases, demonstrating its versatility and practicality:

Static website deployment	Capsium packages can be used to deploy static websites, including HTML, CSS, JS, and media files, on cloud file hosting services.
Microservices integration	Capsium packages can mount routes from other deployable objects, facilitating the integration of microservices.
Data migration	Capsium packages can be used to securely and efficiently migrate data across different platforms and environments.

Secure interchange	Capsium packages provide a secure method for exchanging data between systems, with support for encryption and digital signatures.
Version control and updates	Capsium packages support versioning, enabling efficient updates and rollbacks of deployed static websites.

These use cases highlight the practical applications of the Capsium framework in various scenarios, emphasizing its ability to enhance the deployment and management of web-compatible data packages.

5. Capsium package

The Capsium package is the fundamental unit of deployment within the Capsium framework. This clause details the structure, contents, and specifications of a Capsium package, ensuring a comprehensive understanding of its components and functionality.

5.1. General

A Capsium package is a compressed, single-file deployable object that contains all the necessary files for a static site. It is designed to be easily built, inspected, extracted, and deployed across various platforms. The package ensures that all resources, metadata, and configurations required for site deployment are self-contained within it.

Description	A Capsium package encapsulates a static website, including HTML, CSS, JS, media files, and potentially a data store.
MIME type	The MIME type for a Capsium package is `application/vnd.capsium.package`.
File extension	The standard file extension for a Capsium package is `.cap`.

5.2. Structure

5.2.1. General

The structure of a Capsium package includes several key elements that organize and define the contents and functionality of the package:

Folder hierarchy	The internal file/folder hierarchy represents how files will be served by the package's external API.
Metadata, versions, package dependencies	Metadata provides information about the package, including versioning details and dependencies on other Capsium packages.
License and copyright file and declaration	The package includes a license and copyright file, typically in SPDX format, to specify the legal terms of use.

5.2.2. Folder hierarchy

The folder hierarchy is:

```
example-capsium-package/
├── index.html
├── styles.css
```

```
├── app.js
├── manifest.json
├── routes.json
├── http-api.json
├── storage.json
├── security.json
├── authentication.json
├── logging-monitoring.json
├── validation.json
├── LICENSE.spdx
└── README.md
```

Figure 1

5.3. Metadata file

5.3.1. General

```
{
    "name": "example-capsium-package",
    "version": "1.0.0",
    "description": "A sample Capsium package that demonstrates resource bundling.
",
 "guid": "example.com/example-capsium-package",
    "uuid": "123e4567-e89b-12d3-a456-426614174000",
    "author": "Your Name",
    "repository": {
        "type": "git",
        "url": "https://github.com/yourusername/example-capsium-package.git"
    },
    "dependencies": {
        "other-package.capsium": ">=1.0.0"
    },
    "license": "path/to/LICENSE.spdx",
    "readOnly": true,
    "modules": ["signatures", "layered-storage"]
}
```

Figure 2

1) **`name`**
   Description The name of the package.
   Requirements
   — Should be a string.
   — Must be unique within the ecosystem.
   — Typically uses kebab-case (lowercase letters with hyphens).
2) **`version`**
   Description The version of the package.
   Requirements
   — Should follow [Semantic Versioning](https://semver.org/) (e.g., `1.0.0`).
   — Consists of three digits separated by dots, representing major, minor, and patch versions.
3) **`description`**
   Description A brief description of the package.
   Requirements
   — Should be a string.
   — Provides a concise overview of what the package does.
4) **`guid`**
   Description A globally unique identifier for the package.
   Requirements
   — Should be a URI.
   — URI format
5) **`uuid`**
   Description A universally unique identifier for the package.

Requirements
- Should be a string.
- Must be a valid UUID (e.g., `123e4567-e89b-12d3-a456-426614174000`).

6) **`author`**
   Description: The name of the author or maintainer of the package.
   Requirements:
   - Should be a string.
   - Can include the author's name or organization.

7) **`license`**
   Description: The license under which the package is distributed.
   Requirements:
   - Should be a string.
   - Can be a standard license identifier (e.g., `MIT`) or a path to a license file (e.g., `path/to/LICENSE.spdx`).

8) **`repository`**
   Description: Information about the repository where the package source code is hosted.
   Sub-attributes:
   - `type` The type of version control system (e.g., `git`).
     Requirements: Should be a string.
   - `url` The URL of the repository.
     Requirements: Should be a string and a valid URL.

9) **`dependencies`**
   Description: A list of other packages that this package depends on.
   Requirements:
   - Should be an object where keys are package names and values are version requirements.
   - Version requirements can use semantic versioning ranges (e.g., `>=1.0.0`).

10) **readOnly**:
    Type: `boolean`
    Description: Specifies if the package is immutable.
    Value Requirements: Must be set to `true` to activate immutability.
    Example

11) **`modules`** (optional)
    Description: A list of Capsium modules ([modules]) that the package claims conformance to or requires.
    Requirements:
    - Should be an array of strings.
    - Each string should be the identifier of a module defined in [modules] (e.g., `signatures`, `layered-storage`).
    Example

```
"modules": ["signatures", "layered-storage"]
```

Figure 3

NOTE 1 The `name`, `version`, `guid`, and `uuid` attributes are critical for the unique identification of the package.

NOTE 2 The `repository` and `dependencies` attributes help in maintaining and managing the package's source code and its dependencies, respectively.

5.3.2. Identifier

The GUID (Globally Unique Identifier) for a Capsium package is used for uniquely identifying the package within the ecosystem. It follows a URI (Uniform Resource Identifier) format to ensure global uniqueness and to provide a standardized way of referencing the package.

It ensures global uniqueness, readability, and consistent identification of packages.

The GUID is essential for dependency tracking, providing a reliable reference to specific packages within the ecosystem.

5.3.2.1. Requirements for GUID in URI Format

1) Structure:
   — The GUID should be structured in a URI format.
   — Typically, it follows a reverse domain name notation to ensure uniqueness.
2) Components:
   Scheme The scheme part of the URI, which could be `http`, `https`, or a custom scheme like `capsium`.
   Authority This usually includes the domain name, ensuring the identifier is unique to an organization or individual.
   Path A path that typically reflects the package name and possibly the version.
3) Uniqueness:
   — The GUID must be unique across all packages to avoid conflicts.
   — Using the domain name owned by the package maintainer helps ensure uniqueness.
4) Readability:
   — The GUID should be easy to read and understand, reflecting the package's origin and name.
5) Examples:
   — The GUID should ideally be in lowercase to maintain consistency and avoid case-sensitivity issues.

5.3.2.2. Examples of GUIDs in URI Format

Here are a few examples of GUIDs that comply with the URI format requirements:

1) Example 1:

```
"guid": "capsium://example.com/package-name"
```

Figure 4

Scheme	`capsium`
Authority	`example.com`
Path	`/package-name`

1) Example 2:

```
"guid": "https://example.com/packages/sample-package"
```

Figure 5

Scheme	`https`
Authority	`example.com`
Path	`/packages/sample-package`

1) Example 3:

```
"guid": "http://myorganization.org/capsium/my-package"
```

Figure 6

Scheme	`http`

Authority	`myorganization.org`
Path	`/capsium/my-package`

1) Example 4:

```
"guid": "capsium://opensource.org/libs/lib-capsium"
```

Figure 7

Scheme	`capsium`
Authority	`opensource.org`
Path	`/libs/lib-capsium`

5.3.2.3. Dependency tracking

The GUID is crucial for dependency tracking as it provides a unique and consistent identifier for each package. When defining dependencies in the `metadata.json` file, the GUID ensures that the correct package is referenced, avoiding confusion with similarly named packages.

In the `metadata.json` file, dependencies can be listed using the GUID:

```
{
    "dependencies": {
        "capsium://example.com/package-name": ">=1.0.0",
        "https://example.com/packages/another-package": "^2.1.0"
    }
}
```

Figure 8

`capsium://example.com/package-name`	This GUID uniquely identifies the `package-name` from `example.com` and specifies that any version `>=1.0.0` is acceptable.
`https://example.com/packages/another-package`	This GUID uniquely identifies the `another-package` from `example.com` and specifies that any version compatible with `2.1.0` (using semantic versioning) is acceptable.

5.3.3. Versions

5.3.3.1. General

Versions in the metadata file use [Semantic Versioning](https://semver.org/), which follows the `MAJOR.MINOR.PATCH` format.

Here is an example:

```
{
    "version": "1.0.0"
}
```

Figure 9

The version of a Capsium package is a critical attribute that indicates the state and compatibility of the package over time. It follows the Semantic Versioning (SemVer) convention to ensure clarity and consistency across package versions.

5.3.3.2. Requirements for Version

1) Format:
   — The version should follow the Semantic Versioning format: `MAJOR.MINOR.PATCH`.
   — Each component (MAJOR, MINOR, PATCH) should be a non-negative integer without leading zeros.
2) Components:

MAJOR	Incremented for incompatible API changes. When you make changes that break backward compatibility, you increase the major version.
MINOR	Incremented for adding functionality in a backward-compatible manner. When you add new features that do not break existing functionality, you increase the minor version.
PATCH	Incremented for backward-compatible bug fixes. When you make minor changes or fixes that do not affect the API, you increase the patch version.

1) **Pre-release and Build Metadata** (Optional):

Pre-release version	Indicated by appending a hyphen and a series of dot-separated identifiers (e.g., `1.0.0-alpha`, `1.0.0-beta.1`).
Build metadata	Indicated by appending a plus sign and a series of dot-separated identifiers (e.g., `1.0.0+20130313144700`, `1.0.0-beta+exp.sha.5114f85`).

3) Incrementing Versions:
   — Always increment the appropriate part of the version number based on the nature of the changes.
   — Reset the lower components to zero when incrementing a higher component (e.g., `1.2.3` to `2.0.0`).
4) Uniqueness:
   — Each release of a package should have a unique version number to distinguish it from other releases.

5.3.3.3. Examples of Version Numbers

1) Stable Versions:
   — `1.0.0`: Initial stable release.
   — `2.1.0`: Minor update with new features that are backward-compatible.
   — `3.0.2`: Patch update with bug fixes for the third major version.
2) Pre-release Versions:
   — `1.0.0-alpha`: An alpha version, which is an early release not intended for production use.
   — `1.0.0-beta.1`: The first beta release, which is more stable than alpha but still not production-ready.
   — `1.0.0-rc.1`: The first release candidate, which is a final stage before a stable release.
3) Versions with Build Metadata:
   — `1.0.0+20130313144700`: A stable release with build metadata indicating the build timestamp.
   — `2.0.0-beta+exp.sha.5114f85`: A beta release with experimental build metadata.

5.3.4. Dependencies

5.3.4.1. General

Dependencies are specified in the `dependencies` section of the metadata JSON file. Each dependency is listed with a name and a version requirement.

The `dependencies` section in the `metadata.json` file specifies other packages that the Capsium package depends on. This section ensures that all necessary packages are available for the

package to function correctly. Dependency resolution and the bundling of dependent packages into composite packages are defined by the composite packages module ([module-composite]).

Here is an example:

```
{
    "dependencies": {
        "other-package.capsium": ">=1.0.0",
        "another-package.capsium": "^2.3.4"
    }
}
```

Figure 10

5.3.4.2. Requirements for Dependencies

1) Structure:
   — The `dependencies` section should be an object where each key is the GUID of a dependency package and the corresponding value is the version requirement.
2) GUID:
   — The key should be the GUID of the dependency package in URI format, ensuring global uniqueness and proper identification.
3) Version Requirement:
   — The value should be a string that specifies the version requirement of the dependency.
   — Version requirements can use semantic versioning ranges, such as:
      — Exact version: `1.2.3`
      — Greater than or equal to a version: `>=1.0.0`
      — Compatible with a version: `^2.1.0`
      — Ranges: `>=1.0.0 <2.0.0`
4) Multiple Dependencies:
   — The `dependencies` section can list multiple dependencies, each with its GUID and version requirement.

5.3.4.3. Examples of Dependencies

1) Single Dependency:

```
"dependencies": {
    "capsium://example.com/package-name": ">=1.0.0"
}
```

Figure 11

— This specifies that the package depends on `package-name` from `example.com` with any version `>=1.0.0`.
   1) Multiple Dependencies:

```
"dependencies": {
    "https://example.com/packages/first-package": "^2.1.0",
    "capsium://another.com/second-package": "1.2.3"
}
```

Figure 12

— This specifies that the package depends on:
   — `first-package` from `example.com` with any version compatible with `2.1.0`.
   — `second-package` from `another.com` with the exact version `1.2.3`.

1) Range Version Dependency:

```
"dependencies": {
    "capsium://example.org/dependency-package": ">=1.0.0 <2.0.0"
}
```

Figure 13

— This specifies that the package depends on `dependency-package` from `example.org` with any version between `1.0.0` (inclusive) and `2.0.0` (exclusive).

1) Pre-release Version Dependency:

```
"dependencies": {
    "capsium://example.net/experimental-package": "1.0.0-beta.1"
}
```

Figure 14

— This specifies that the package depends on `experimental-package` from `example.net` with the specific pre-release version `1.0.0-beta.1`.

1) Dependency with Build Metadata:

```
"dependencies": {
    "https://example.com/special-package": "1.0.0+20130313144700"
}
```

Figure 15

— This specifies that the package depends on `special-package` from `example.com` with the exact version `1.0.0` including build metadata `20130313144700`.

1) Multiple Version Ranges:

```
"dependencies": {
    "capsium://example.org/multi-range-package": ">=1.0.0 <1.5.0 || >=2.0.0
<3.0.0"
}
```

Figure 16

— This specifies that the package depends on `multi-range-package` from `example.org` with versions either between `1.0.0` (inclusive) and `1.5.0` (exclusive) or between `2.0.0` (inclusive) and `3.0.0` (exclusive).

1) Wildcard Version Dependency:

```
"dependencies": {
    "capsium://example.com/wildcard-package": "*"
}
```

Figure 17

— This specifies that the package depends on `wildcard-package` from `example.com` with any available version.

1) Caret (^) and Tilde (~) Ranges:

```
"dependencies": {
```



```
    "capsium://example.com/caret-package": "^1.2.3",
    "capsium://example.com/tilde-package": "~1.2.3"
}
```

Figure 18

— This specifies that the package depends on:
  — `caret-package` from `example.com` with any version compatible with `1.2.3` (meaning `>=1.2.3 <2.0.0`).
  — `tilde-package` from `example.com` with any version compatible with `1.2.3` (meaning `>=1.2.3 <1.3.0`).

Each dependency in the `dependencies` section ensures that the package has access to the required versions of other packages necessary for its proper functionality.

5.3.5. License

5.3.5.1. General

The `license` key in the `metadata.json` file specifies the licenses under which the Capsium package is distributed. This key ensures compliance with legal requirements and informs users of their rights and obligations regarding the package.

The license file should be in the SPDX format and referenced from the metadata file.

5.3.5.2. Requirements for License

1) Format:
   — The `license` key should be a string or an array of objects.
   — Each string should be a valid SPDX (Software Package Data Exchange) license identifier or a path to an SPDX file included in the package.
2) Single License:
   — When the package is distributed under a single license, the `license` key should be a string.
3) Multiple Licenses:
   — When the package is distributed under multiple licenses, the `license` key should be an array of objects.
   — Each object in the array should specify a `type` and an optional `file` field if pointing to an SPDX file.
   — Each object should also include a `condition` field that describes when the license applies.
4) SPDX Identifier or File:
   — An SPDX identifier should be a valid SPDX license identifier.
   — An SPDX file should be a path to a file included in the package that contains the SPDX license text.

5.3.5.3. Examples of License

1) Single SPDX License:

```
"license": "MIT"
```

Figure 19

— This specifies that the package is distributed under the MIT License.
  1) Single SPDX File License:

```
"license": "LICENSE.spdx"
```

Figure 20

— This specifies that the package is distributed under the license detailed in the `LICENSE.spdx` file.

1) Multiple Licenses with Conditions:

```
"license": [
    {
        "type": "MIT",
        "condition": "Default license"
    },
    {
        "type": "Apache-2.0",
        "condition": "For use in commercial environments"
    }
]
```

Figure 21

— This specifies that the package is distributed under the MIT License by default, but under the Apache License 2.0 when used in commercial environments.

1) Combination of SPDX Identifier and File with Conditions:

```
"license": [
    {
        "type": "MIT",
        "condition": "Default license"
    },
    {
        "type": "Custom-License",
        "file": "custom-license.spdx",
        "condition": "For internal use only"
    }
]
```

Figure 22

— This specifies that the package is distributed under the MIT License by default, but under a custom license detailed in the `custom-license.spdx` file for internal use only.

1) Complex License Conditions:

```
"license": [
    {
        "type": "GPL-3.0-only",
        "condition": "When redistributed"
    },
    {
        "type": "LGPL-3.0-only",
        "condition": "When used as a library"
    }
]
```

Figure 23

— This specifies that the package is distributed under the GPL-3.0-only License when redistributed and under the LGPL-3.0-only License when used as a library.

By following these requirements and examples, the `license` key in the Capsium package's `metadata.json` file provides clear information about the applicable licenses and the conditions under which they apply.

Below is an example of a simple SPDX license file (`LICENSE.spdx`):

```
SPDXVersion: SPDX-2.1
DataLicense: CC0-1.0
SPDXID: SPDXRef-DOCUMENT
DocumentName: example-capsium-package
DocumentNamespace: http://spdx.org/spdxdocs/example-capsium-package-abc123
Creator: Person: John Doe
Creator: Organization: Example Organization
Creator: Tool: SPDX-Tools-Version-2.1.0
Created: 2024-05-28T12:00:00Z
LicenseID: MIT
LicenseName: MIT License
LicenseText: |
    MIT License

    Permission is hereby granted, free of charge, to any person obtaining a copy
    of this software and associated documentation files (the "Software"), to deal
    in the Software without restriction, including without limitation the rights
    to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
    copies of the Software, and to permit persons to whom the Software is
    furnished to do so, subject to the following conditions:

    The above copyright notice and this permission notice shall be included in
all
    copies or substantial portions of the Software.

    THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
    IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
    FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
    AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
    LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
    OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
    SOFTWARE.
```

Figure 24

Ensure this SPDX license file is referenced in the `manifest.json`:

```
{
    "license": "path/to/LICENSE.spdx"
}
```

Figure 25

5.3.6. Read-only

Capsium packages can be configured as immutable, ensuring that their content cannot be modified after creation. Composite packages layer changes on top of immutable base packages ([module-composite]).

This section details the requirements, specifications, and use cases for configuring a package as read-only, including value requirements and enumerations for attributes. The read-only attribute is package-wide and set inside `metadata.json`.

5.4. Manifest file

5.4.1. General

The manifest file describes how to handle multi-format content within the package.

It includes mappings and configurations for handling different types of files and resources.

When the `manifest.json` file does not exist, it should be built automatically from the contents of the `contents/` directory.

Specification	File name	`manifest.json`
	Location	Root directory of the package
	Content	JSON format, specifying the resources, their versions, and configurations.

Example:

```
{
    "resources": {
        "index.html": {
            "type": "text/html",
            "version": "1.0.0"
        },
        "styles.css": {
            "type": "text/css",
            "version": "1.0.0"
        },
        "app.js": {
            "type": "application/javascript",
            "version": "1.0.0"
        },
        "dynamic-content.js": {
            "type": "application/javascript",
            "version": "1.0.0"
        },
        "mobile.css": {
            "type": "text/css",
            "version": "1.0.0"
        },
        "desktop.css": {
            "type": "text/css",
            "version": "1.0.0"
        },
        "images/small.jpg": {
            "type": "image/jpeg",
            "version": "1.0.0"
        },
        "images/medium.jpg": {
            "type": "image/jpeg",
            "version": "1.0.0"
        },
        "images/large.jpg": {
```



```
            "type": "image/jpeg",
            "version": "1.0.0"
        },
        "content/en/index.html": {
            "type": "text/html",
            "version": "1.0.0"
        },
        "content/en/about.html": {
            "type": "text/html",
            "version": "1.0.0"
        },
        "content/es/index.html": {
            "type": "text/html",
            "version": "1.0.0"
        },
        "content/es/about.html": {
            "type": "text/html",
            "version": "1.0.0"
        }
    }
}
```

Figure 26

5.4.2. Content visibility

Content visibility in the Capsium package is managed through the `manifest.json` file, where resources can be designated as either `exported` or `private`. This designation determines whether the resource can be re-used by other packages or is restricted to the current package. Re-use of `exported` resources across packages is defined by the composite packages module ([module-composite]).

Requirements and Specifications

1) Resource Declaration:
   - Resources must be declared in the `manifest.json` file.
   - Each resource entry should include the path to the resource and its visibility status.
2) Visibility Options:
   - Exported: Resources marked as `exported` are available for re-use by other packages that depend on the current package.
   - Private: Resources marked as `private` are restricted to the current package and cannot be accessed by other packages.
3) Example Configuration:
   - An example `manifest.json` file demonstrating resource visibility:

```
{
    "resources": [
        {
            "path": "scripts/main.js",
            "visibility": "exported"
        },
        {
            "path": "styles/theme.css",
            "visibility": "private"
        },
        {
            "path": "images/logo.png",
            "visibility": "exported"
```

```
            }
        ]
    }
```

Figure 27

1) Usage in Dependent Packages:
   — Packages that depend on another package can access resources marked as `exported` by including the appropriate references in their own configuration files.
   — Example usage in a dependent package — a dependency resource reference of the form `<guid>/<path>` ([module-composite]):

```
{
    "dependencies": {
        "capsium://example.com/capsium-core": "1.0.0"
    },
    "routes": [
        {
            "path": "/vendor/core/main.js",
            "resource": "capsium://example.com/capsium-core/scripts/main.js"
        }
    ]
}
```

Figure 28

1) Enforcement:
   — The Capsium system should enforce visibility rules, ensuring that private resources are not accessible to other packages.
   — Attempts to access private resources from other packages should result in an error, maintaining the integrity of resource boundaries.

By clearly defining and adhering to these visibility rules, the Capsium package ensures that resource bundling is both flexible and secure, allowing for effective re-use of assets while protecting private resources.

5.5. Root file

The root file of a Capsium package serves as the main entry point and must be an HTML file. This file is crucial as it defines the primary structure and content that the system should load or render.

5.5.1. Requirements for Root File

1) File Type:
   — The root file must be an HTML file. It should have an `.html` extension.
2) File Location:
   — The root file should be located within the package directory.
   — The path to the root file should be specified relative to the root directory of the package.
3) File Naming:
   — The root file should have a clear and descriptive name, commonly named `index.html` or `main.html`.
4) Entry Point Specification:
   — The path to the root HTML file should be accurately specified in the `index` key of the `routes.json` file.
   — Ensure the path does not contain typos or incorrect directory names.
5) Content Requirements:
   — The HTML file should include the necessary structure (`<html>`, `<head>`, and `<body>` tags).
   — It must be well-formed and valid HTML to ensure proper rendering and functionality.

5.5.2. Examples of Root File

1) Basic HTML Entry Point:

```
{
    "index": "public/index.html"
}
```

Figure 29

— This specifies that the root file for the package is `index.html` located in the `public` directory.
1) HTML Entry Point in Documentation Directory:

```
{
    "index": "docs/main.html"
}
```

Figure 30

— This specifies that the root file for the package is `main.html` located in the `docs` directory.
1) HTML Entry Point in Web Directory:

```
{
    "index": "web/index.html"
}
```

Figure 31

— This specifies that the root file for the package is `index.html` located in the `web` directory.
1) HTML Entry Point in Dist Directory:

```
{
    "index": "dist/index.html"
}
```

Figure 32

— This specifies that the root file for the package is `index.html` located in the `dist` directory.
1) HTML Entry Point in Root Directory:

```
{
    "index": "index.html"
}
```

Figure 33

— This specifies that the root file for the package is `index.html` located in the root directory of the package.

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Capsium Package</title>
```

```
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <h1>Welcome to the Capsium Package</h1>
    <script src="app.js"></script>
</body>
</html>
```

Figure 34

5.6. Resource bundling

5.6.1. General

Resource bundling involves packaging all necessary static and dynamic content within the Capsium package to ensure that the package is self-contained and can be served efficiently.

The content include:

Static content	Includes HTML, CSS, JS, images, and other media files.
Dynamic content	Although primarily static, the package can include references to dynamic content handled by client-side scripts.
Conditional alternative content	The package can include alternative content that is conditionally loaded based on specific criteria, such as files with alternative image resolutions.

Example directory structure:

```
example-capsium-package/
├── index.html
├── styles.css
├── app.js
├── dynamic-content.js
├── mobile.css
├── desktop.css
├── images/
│   ├── small.jpg
│   ├── medium.jpg
│   └── large.jpg
├── content/
│   ├── en/
│   │   ├── index.html
│   │   └── about.html
│   ├── es/
│   │   ├── index.html
│   │   └── about.html
├── metadata.json
├── manifest.json
├── routes.json
├── LICENSE.spdx
└── README.md
```

Figure 35

5.6.2. Static Content

Static content includes files that do not change once the package is created and can be directly served to the client. These files are essential for the visual and functional aspects of the web application.

HTML Files

These files define the structure and layout of web pages. They typically include elements like headers, paragraphs, links, and embedded resources such as images and scripts.

Example:

Figure 36

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Example Page</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <h1>Welcome to Example Page</h1>
    <script src="app.js"></script>
</body>
</html>
```

Figure 37

CSS Files

These files define the styles for HTML elements, specifying colors, fonts, layouts, and other visual aspects.

Example (pass:c,q,a,m,p[`styles.css`]):

Figure 38

```
body {
    font-family: Arial, sans-serif;
    background-color: =f0f0f0;
}
h1 {
    color: =333;
}
```

Figure 39

JavaScript Files

These files contain client-side scripts that add interactivity and dynamic behavior to the web pages.

Example (pass:c,q,a,m,p[`app.js`]):

Figure 40

```
document.addEventListener('DOMContentLoaded', () => {
    console.log('Page loaded');
});
```

Figure 41

Images and Media Files
: These include JPEG, PNG, GIF images, SVG graphics, and other media files like videos and audio clips that are used within the web pages.

  Example:

Figure 42

```
example-capsium-package/
├── images/
│   ├── logo.png
│   └── banner.jpg
```

Figure 43

5.6.3. Dynamic Content

Dynamic content refers to content that can change or be generated on the fly, typically handled by client-side scripts. While the package itself is primarily static, it can include references to dynamic content.

Client-side Scripts
: JavaScript files that fetch and display dynamic content from APIs or other sources at runtime.

  Example (pass:c,q,a,m,p[`dynamic-content.js`]):

Figure 44

```
  fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(data => {
          document.getElementById('dynamic-
content').innerText = data.message;
      });
```

Figure 45

Dynamic References
: Links and scripts that point to external resources or APIs that provide dynamic data.

  Example (pass:c,q,a,m,p[`index.html`]):

Figure 46

```
  <div id="dynamic-content"></div>
  <script src="dynamic-content.js"></script>
```

Figure 47

5.6.4. Conditional Alternative Content

Conditional alternative content allows the package to include multiple versions of a resource, with the appropriate version being loaded based on specific criteria. This can enhance performance and provide a better user experience.

Alternative Image Resolutions
: Including images in multiple resolutions and loading the appropriate one based on the device's screen resolution.

Example (pass:c,q,a,m,p[`index.html`]):

Figure 48

```
  <img srcset="small.jpg 480w, medium.jpg 768w, large.
jpg 1024w"
       sizes="(max-width: 600px) 480px, (max-width:
900px) 768px, 1024px"
       src="small.jpg" alt="Example Image">
```

Figure 49

Content for Different Languages

Providing content in multiple languages and loading the appropriate version based on the user's language preferences.

Example:

Figure 50

```
example-capsium-package/
├── content/
│   ├── en/
│   │   ├── index.html
│   │   └── about.html
│   ├── es/
│   │   ├── index.html
│   │   └── about.html
```

Figure 51

JavaScript to load language-specific content (pass:c,q,a,m,p[`language-loader.js`]):

Figure 52

```
  const userLang = navigator.language ||
navigator.userLanguage;
  const contentPath = userLang.startsWith('es') ?
'content/es/' : 'content/en/';
  fetch(contentPath + 'index.html')
      .then(response => response.text())
      .then(html => {
          document.body.innerHTML = html;
      });
```

Figure 53

Device-Specific Content

Serving different versions of content based on the type of device (e.g., mobile vs. desktop).

Example:

Figure 54

```
<link rel="stylesheet" media="screen and (max-width:
600px)" href="mobile.css">
<link rel="stylesheet" media="screen and (min-width:
601px)" href="desktop.css">
```

Figure 55

5.7. Resource routing

5.7.1. General

Resource routing defines how requests to the Capsium package are handled and routed.

The `routes.json` file is the central configuration for routing within your package. It maps URL paths to resources, covering static resource routing and the essential configurations for headers and HTTP methods. Dynamic HTTP API routes that map a URL path and method to an executable handler are defined by the handler routes module ([module-handler-routes]).

The routing file `routes.json` is the single source of truth for routing in the package, simplifying management and ensuring consistency.

When the `routes.json` file doesn't exist, automatically generate it based on the contents in the manifest. When it is an HTML file, create 2 routes, one with the file's base name, one with the full file name. When a file of another type, generate the route relative from the content/ path.

Example `routes.json`

```
{
    "index": "index.html",
    "routes": [
        {
            "path": "/",
            "resource": "index.html"
        },
        {
            "path": "/styles.css",
            "resource": "styles.css"
        },
        {
            "path": "/app.js",
            "resource": "app.js"
        },
        {
            "path": "/images/small.jpg",
            "resource": "images/small.jpg"
        },
        {
            "path": "/data/users",
            "resource": "data/users.json"
        },
        {
            "path": "/data/products",
            "resource": "data/products.json"
        }
    ]
}
```

Figure 56

In this example, the `routes.json` file includes routes for static resources (like HTML, CSS, images, and data files).

5.7.2. Index route

The `index` key in the `routes.json` file designates the entry point or root file of the Capsium package. This key is crucial for defining the primary file that the system should load or execute.

5.7.2.1. Requirements for index Key

1) File Path:
   — The `index` key should be a string representing the relative path to the root file from the root directory of the package.
   — The path should be valid and point to an existing file within the package.
2) File Type:
   — The root file can be of various types depending on the nature of the package (e.g., JavaScript, HTML, JSON). Ensure the file type is appropriate for the package's purpose.
3) Uniqueness:
   — There should be only one `index` key in the `routes.json` file, specifying a single root file.
4) Consistency:
   — The path specified by the `index` key should be consistent with the project's structure and should not include typos or incorrect directory names.

5.7.2.2. Examples of index Key

HTML Entry Point

```
{
    "index": "public/index.html"
}
```

Figure 57

— This specifies that the root file for the package is `index.html` located in the `public` directory.

By adhering to these requirements and examples, the `index` key in the `routes.json` file ensures that the Capsium package has a clearly defined entry point, facilitating proper loading and execution of the package.

5.7.3. Dataset routes

Mounting routes to data sets involves configuring specific endpoints that provide access to various data sets within the package. This allows for organized and efficient data retrieval, enabling users to access the data they need through well-defined routes. All data route mounts will use the HTTP path mount point `/api/v1/data/` as the root.

Route definition	Specifies the URL path that will be used to access the data set, using `/api/v1/data/` as the root.
Data source	Defines the source of the data, such as a file path, database query, or external API.
Response format	Specifies the format in which the data will be returned, such as JSON, XML, or CSV.

Access control for dataset routes is an optional capability defined by the authentication module ([module-authentication]).

Dataset routes should be mounted as specified in `routes.json`, with each route pointing to a key `dataset` that is provided in `storage.json`.

Example of `storage.json`:

```
{
    "storage": {
        "dataSets": {
            "users": {
```

```
                "source": "db/users",
            },
            "products": {
                "source": "files/products.json",
            },
            "sales": {
                "source": "api/external/sales",
            }
        }
    }
}
```

Figure 58

Example of `routes.json`:

```
{
    "routes": [
        {
            "route": "/api/v1/data/users",
            "dataset": "users"
        },
        {
            "route": "/api/v1/data/products",
            "dataset": "products"
        },
        {
            "route": "/api/v1/data/sales",
            "dataset": "sales"
        }
    ]
}
```

Figure 59

In this example, `routes.json` defines three routes, each pointing to a data set specified in `storage.json`:

- Users data set
  - Route: `/api/v1/data/users`
  - Dataset: `users` (refers to the `users` key in `storage.json`)
- Products data set
  - Route: `/api/v1/data/products`
  - Dataset: `products` (refers to the `products` key in `storage.json`)
- Sales data set
  - Route: `/api/v1/data/sales`
  - Dataset: `sales` (refers to the `sales` key in `storage.json`)

5.7.4. Attributes summary

Table 1 — Table 1: Storage Attributes

Attribute	Description
Storage	The root object for storage configuration.

Table 2 — Table 2: DataSets Attributes (in `storage.json`)

Attribute	Description
Source	The source of the data (e.g., database path, file path, external API URL).
Response format	The format in which the data will be returned (e.g., `json`, `xml`, `csv`).

Table 3 — Table 3: Routes Attributes (in `routes.json`)

Attribute	Description
Route	The URL path for accessing the data set, starting with `/api/v1/data/`.
Dataset	The key in `storage.json` that this route points to.

Access control attributes for dataset routes are defined by the authentication module ([[module-authentication]]).

By configuring these attributes, Capsium packages can effectively manage data storage and provide structured access to data sets through defined routes.

5.7.5. Header responses

5.7.5.1. General

In the Capsium package, resource routing allows you to map URLs to specific resources and define how they should be handled. One crucial aspect of resource routing is defining header responses, which can be done directly in the `routes.json` file or via external files.

5.7.5.2. Inline declarations

You can specify header responses directly within the `routes.json` file. This approach embeds the header definitions within the routing configuration, making it straightforward to manage.

Example:

```
{
    "routes": {
        "/api/resource": {
            "GET": {
                "file": "handlers/getResource.js",
                "headers": {
                    "Content-Type": "application/json",
                    "Cache-Control": "no-cache",
                    "Access-Control-Allow-Origin": "*"
                }
            },
            "POST": {
                "file": "handlers/postResource.js",
                "headers": {
                    "Content-Type": "application/json",
                    "Access-Control-Allow-Origin": "*"
                }
            }
        }
    }
}
```

Figure 60

In this example: - The `GET` method for `/api/resource` has headers defined directly in the `routes.json` file. - The `POST` method for `/api/resource` also defines its headers directly.

5.7.5.3. Declaring through external files

Alternatively, you can manage header definitions in external files, which can be useful for maintaining cleaner and more modular configurations.

Example:

```
{
    "routes": {
        "/api/resource": {
            "GET": {
                "file": "handlers/getResource.js",
                "headersFile": "headers/getResourceHeaders.json"
            },
            "POST": {
                "file": "handlers/postResource.js",
                "headersFile": "headers/postResourceHeaders.json"
            }
        }
    }
}
```

Figure 61

In this example: - The `GET` method for `/api/resource` references an external file `headers/getResourceHeaders.json` for headers. - The `POST` method for `/api/resource` references an external file `headers/postResourceHeaders.json` for headers.

The content of `headers/getResourceHeaders.json` might look like this:

```
{
    "Content-Type": "application/json",
    "Cache-Control": "no-cache",
    "Access-Control-Allow-Origin": "*"
}
```

Figure 62

And `headers/postResourceHeaders.json` might look like this:

```
{
    "Content-Type": "application/json",
    "Access-Control-Allow-Origin": "*"
}
```

Figure 63

By using these mechanisms, you can effectively manage and define header responses in the Capsium package, either directly within the `routes.json` file or through external files for better modularity and maintainability.

5.7.6. Route visibility

Route visibility in the Capsium package is managed through the `routes.json` file, where routes can be designated as either `exported` or `private`. This designation determines whether the route can be re-used by other packages or is restricted to the current package. Re-use of `exported` routes across packages is defined by the composite packages module ([module-composite]).

Requirements and Specifications	1) Route Declaration: — Routes must be declared in the `routes.json` file. — Each route entry should include the path to the resource and its visibility status. 2) Visibility Options: Exported Routes marked as `exported` are available for re-use by other packages that depend on the current package.

Private — Routes marked as `private` are restricted to the current package and cannot be accessed by other packages.

3) Example Configuration:
   — An example `routes.json` file demonstrating route visibility:

```
{
    "routes": [
        {
            "path": "/api/public",
            "handler": "publicHandler",
            "visibility": "exported"
        },
        {
            "path": "/api/private",
            "handler": "privateHandler",
            "visibility": "private"
        }
    ]
}
```

Figure 64

Inheriting and processing routes from dependency packages is an optional capability defined by the composite packages module ([module-composite]).

5.8. Storage

The Capsium package includes a comprehensive storage system that supports static data files, databases, and — through the optional layered storage capability ([module-layered-storage]) — a unified merged filesystem view. This section outlines the requirements and specifications for each storage type.

A Capsium package can contain datasets. Each dataset is composed of data items. A dataset can be one of:

Layered storage — Utilizes a unified merged filesystem view, similar to overlay FS, to manage different layers of content ([module-layered-storage]).

Static File with Structured Data Backed by Schemas — Formats such as CSV, JSON, YAML. Formats that use JSON Schema or YAML Schema for validation.

Static data files — Contains immutable data files that are part of the package.

Databases — Includes support for embedded databases such as SQLite to store structured data (if supported by the Capsium reactor).

5.8.1. Defining datasets

Datasets are explicitly defined, and are referred by other components, such as Clause 5.7.3.

source — Defines the source of the data, such as a file path, database query, or external API.

Example:

```
{
```

```
    "storage": {
        "dataSets": {
            "users": {
                "source": "db/users",
            },
            "products": {
                "source": "files/products.json",
            },
            "sales": {
                "source": "api/external/sales",
            }
        }
    }
}
```

Figure 65

5.8.2. Static data files

Static data files are immutable files that are served directly by the Capsium package. These files typically include assets such as images, stylesheets, and JavaScript files.

Requirements and Specifications

1) Storage Location:
   — Static data files must be stored in a designated directory, such as `static` or `public`.
2) Access Path:
   — Static files should be accessible via predictable URL paths, typically mirroring their directory structure.
   — Example: `/static/images/logo.png` should map to `static/images/logo.png` in the filesystem.
3) Caching:
   — Static files should be served with appropriate caching headers to improve performance.
   — Example:

```
Cache-Control: public, max-age=31536000
```

Figure 66

1) Configuration:
   — The location of static files should be defined in the `routes.json` file.
   — Example:

```
{
    "static": {
        "path": "static",
        "url": "/static"
    }
}
```

Figure 67

5.8.3. Datasets

Datasets in the Capsium package provide structured storage for dynamic data that requires querying and transactional operations.

JSON schema or YAML schema for datasets using static files like JSON or YAML in the `storage.json` configuration file.

5.8.3.1. Schema-backed file datasets

When working with datasets in static file formats such as JSON or YAML, it's important to validate the data against a predefined schema. This ensures the data adheres to the expected structure and types. The `storage.json` configuration file can include references to these schemas.

The `storage.json` file should be structured to include the following attributes for each dataset:

datasetId — Identifier for the dataset.

dataFile — Path to the static data file (JSON or YAML).

schemaFile — Path to the schema file (JSON Schema or YAML Schema).

schemaType — Type of the schema (e.g., "json-schema" or "yaml-schema").

Example structure: `json { "datasets": [ { "datasetId": "dataset1", "dataFile": "/path/to/datafile.json", "schemaFile": "/path/to/schemafile.json", "schemaType": "json-schema" }, { "datasetId": "dataset2", "dataFile": "/path/to/datafile.yaml", "schemaFile": "/path/to/schemafile.yaml", "schemaType": "yaml-schema" } ] } `

Here's an example of a JSON schema for validating a dataset of user information: `json { "$schema": "http://json-schema.org/draft-07/schema#", "type": "object", "properties": { "users": { "type": "array", "items": { "type": "object", "properties": { "id": { "type": "string" }, "name": { "type": "string" }, "email": { "type": "string", "format": "email" } }, "required": ["id", "name", "email"] } } }, "required": ["users"] } `

Here's an example of a YAML schema for validating a dataset of user information: `yaml %YAML 1.2 —- $schema: "http://json-schema.org/draft-07/schema#" type: "object" properties: users: type: "array" items: type: "object" properties: id: type: "string" name: type: "string" email: type: "string" format: "email" required: — id — name — email required: — users `

5.8.3.2. SQLite database

In addition to static file formats like JSON and YAML, datasets can also be stored in SQLite databases. This section explains how to configure SQLite datasets in the `storage.json` file.

The `storage.json` file should include the following attributes for each SQLite dataset:

datasetId — Identifier for the dataset.

databaseFile — Path to the SQLite database file.

schemaFile — Path to the schema file (SQL schema or JSON schema for defining table structures).

schemaType — Type of the schema (e.g., "sql-schema" or "json-schema").

table — Name of the table in the SQLite database that the dataset corresponds to.

Example structure: `json { "datasets": [{ "datasetId": "dataset1", "databaseFile": "/path/to/database.sqlite", "schemaFile": "/path/to/schemafile.sql", "schemaType": "sql-schema", "table": "users" }, { "datasetId": "dataset2", "databaseFile": "/path/to/another_database.sqlite", "schemaFile": "/path/to/schemafile.json", "schemaType": "json-schema", "table": "products" }] } `

Here's an example of an SQL schema for a table of user information: `sql CREATE TABLE users ( id TEXT PRIMARY KEY, name TEXT NOT NULL, email TEXT NOT NULL UNIQUE ); `

Here's an example of a JSON schema for validating the structure of a table of user information: `json { "$schema": "http://json-schema.org/draft-07/schema#", "type": "object", "properties": { "id": { "type": "string" }, "name": { "type": "string" }, "email": { "type": "string", "format": "email" } }, "required": ["id", "name", "email"] } `

5.8.3.3. Validation

To validate the datasets in a YAML, JSON dataset or a SQLite database against their respective schemas, you can use various tools and libraries depending on the programming language.

5.8.4. REST API for data access and modification

This is the HTTP REST API that the Capsium reactor offers for an activated Capsium package to the HTTP user. The following endpoints are supported for all datasets, as defined in `storage.json`.
NOTE The dataset CRUD API served by a reactor — mount points under `/api/v1/data/`, item identity, and exact status codes — is defined by the writable packages module ([module-writable-packages]). This section describes the operations abstractly.

5.8.4.1. GET (Fetch Data)

Endpoint	`/dataset/{datasetId}/data`
Method	GET
Description	Fetches all data items from the specified dataset.
	Request Attributes
datasetId	The identifier of the dataset. Must be a string.
	Response
status	200 OK on success.
body	An array of data items.

Example: `json { "datasetId": "dataset1" } `

5.8.4.2. GET (Fetch Single Data Item)

Endpoint	`/dataset/{datasetId}/data/{dataId}`
Method	GET
Description	Fetches a single data item from the specified dataset.
	Request Attributes
datasetId	The identifier of the dataset. Must be a string.
dataId	The identifier of the data item to be fetched. Must be a string.
	Response
status	200 OK on success.
body	The requested data item.

Example: `json { "datasetId": "dataset1", "dataId": "dataItem42" } `

5.8.4.3. POST (Create Data Item)

Endpoint	`/dataset/{datasetId}/data`
Method	POST
Description	Adds a new data item to the specified dataset.
	Request Attributes
datasetId	The identifier of the dataset. Must be a string.
data	The data item to be added. Must conform to the dataset's schema.
	Response
status	201 Created on success.
body	The created data item with its new identifier.

Example: `json { "datasetId": "dataset1", "data": { "key1": "value1", "key2": 123 } } `

5.8.4.4. PUT (Update Data Item)

Endpoint	`/dataset/{datasetId}/data/{dataId}`
Method	PUT
Description	Updates an existing data item in the specified dataset.
	Request Attributes
datasetId	The identifier of the dataset. Must be a string.
dataId	The identifier of the data item to be updated. Must be a string.
data	The updated data item. Must conform to the dataset's schema.
	Response
status	200 OK on success.
body	The updated data item.

Example: `json { "datasetId": "dataset1", "dataId": "dataItem42", "data": { "key1": "newValue", "key2": 456 } } `

5.8.4.5. DELETE (Delete Data Item)

Endpoint	`/dataset/{datasetId}/data/{dataId}`
Method	DELETE
Description	Deletes a data item from the specified dataset.
	Request Attributes
datasetId	The identifier of the dataset. Must be a string.
dataId	The identifier of the data item to be deleted. Must be a string.
	Response

status	204 No Content on success.

Example: `json { “datasetId”: “dataset1”, “dataId”: “dataItem42” } `

5.8.5. Data Persistence

5.8.5.1. General

A Capsium package containing data may allow modification of data inside the included datasets. Configuration needs to be specified in the package on which data files can be modified or updated.
NOTE Reactors claiming the writable packages module implement modification with the reactor-side overlay model — top-layer writes, tombstones, and dataset logs — defined by [module-writable-packages]. The action history mechanism described here is an alternative package-side design for recording the same changes.

Since a Capsium package at its core is immutable, the mechanism for handling modifications is by storing action patches in an “action history” folder. This folder can be external to the Capsium package or inside a composite Capsium package ([module-composite]). Each data change is stored as a separate patch file.

When a Capsium reactor loads a Capsium package with an action history folder, it will replay those actions on top of the dataset so that the changes persist for users who access the activated Capsium package.

5.8.5.2. Action patch

An action patch represents a single change to a dataset. The following attributes are required:

timestamp	The time when the change was made. Must be in ISO 8601 format (e.g., `2024-05-28T12:34:56Z`).
user	The identifier of the user who made the change. Should be a string.
action	The type of action performed. Enumerated values: add, `update`, `delete`.
datasetId	The identifier of the dataset affected by the action. Should be a string.
dataId	The identifier of the data item affected by the action. Should be a string.
changes	A JSON object detailing the changes made. The format depends on the type of action.

Example: `json { “timestamp”: “2024-05-28T12:34:56Z”, “user”: “user123”, “action”: “update”, “datasetId”: “dataset1”, “dataId”: “dataItem42”, “changes”: { “key1”: “newValue” } } `

5.8.5.3. Action history folder

The action history folder stores all action patches. The folder must have the following structure and attributes:

Location	Can be external to the Capsium package or inside a composite Capsium package.
Structure	Each action patch is stored as a separate file within the folder.
Filename Convention	Each file name should be unique and can be based on the timestamp and user ID (e.g., `20240528T123456Z_user123_update_dataItem42.json`).

File Content	Each file must contain a valid action patch JSON object as specified above.

Example: ` action-history/ ├── 20240528T123456Z_user123_update_dataItem42.json ├── 20240529T101112Z_user456_add_dataItem43.json └── ... `

Additional requirements for the action history folder:

Access Control	The folder must be secured to prevent unauthorized access. Only designated users or systems should have read/write access.
Backup	Regular backups of the action history folder should be maintained to prevent data loss.
Versioning	Each action patch should include a version attribute to manage changes to the action patch schema.

Example of an action patch with versioning: `json { “version”: “1.0”, “timestamp”: “2024-05-28T12:34:56Z”, “user”: “user123”, “action”: “update”, “datasetId”: “dataset1”, “dataId”: “dataItem42”, “changes”: { “key1”: “newValue” } } `

5.8.5.4. Saving Data Changes in Datasets to a New External Capsium Package

To save data changes in datasets to a new external Capsium package, the following configuration is required:

Configuration File	`save-external.json`	
Attributes	originalPackageId	The identifier of the original Capsium package. Must be a string.
	newPackageId	The identifier for the new external Capsium package. Must be a string.
	actionHistoryLocation	The location of the action history folder. Must be a valid path.

Example: `json { “originalPackageId”: “capsiumPkg1”, “newPackageId”: “capsiumPkg2”, “actionHistoryLocation”: “/path/to/action-history” } `

The process for saving data changes includes the following steps:

1) Identify Changes: Collect all action patches related to the dataset modifications.
2) Create New Package: Generate a new Capsium package structure.
3) Include Action Patches: Copy the action patches to the new package’s action history folder.
4) Update Metadata: Modify the `storage.json` and other relevant configuration files to reflect the new package and its contents.
5) Validate Package: Ensure that the new package meets all Capsium package requirements and is properly versioned.

Example process: `shell # Collect action patches cp /path/to/action-history/* /new-package/action-history/

6. Create new package structure

mkdir /new-package cp -r /original-package/* /new-package/

7. Update metadata

jq '.packages += [{"id": "capsiumPkg2", "actionHistoryLocation": "/new-package/action-history"}]' /new-package/storage.json > /new-package/storage_tmp.json mv /new-package/storage_tmp.json /new-package/storage.json

8. Validate package

capsium-validate /new-package `

Additional considerations for creating a new external Capsium package:

Integrity Checks	Perform integrity checks to ensure that all data items and action patches are correctly included and no data corruption has occurred.
Documentation	Update the package documentation to reflect the changes and new version information.
Notification	Notify users or systems that depend on the package about the update to the new package.

Example of integrity check: `shell capsium-check-integrity /new-package `

Example of updating documentation: `markdown # Capsium Package Documentation

8.1. Package ID: capsiumPkg2

Description	This package includes updated datasets from capsiumPkg1 with action patches applied.
Version	2.0
Action History Location	/new-package/action-history `

8.1.1. Saving Data Changes in Datasets in a Composite Capsium Package That Contains the Current Package

To save data changes in datasets in a composite Capsium package, the following configuration is required:

Configuration File	`save-composite.json`	
Attributes	originalPackageId	The identifier of the original Capsium package. Must be a string.
	compositePackageId	The identifier for the composite Capsium package. Must be a string.
	internalActionHistoryPath	The path to the action history folder within the composite Capsium package. Must be a valid internal path.

Example: `json { "originalPackageId": "capsiumPkg1", "compositePackageId": "compositeCapsiumPkg1", "internalActionHistoryPath": "internal/action-history" } `

The process for saving data changes in a composite Capsium package includes the following steps:

1) Identify Changes: Collect all action patches related to the dataset modifications.
2) Create Composite Package Structure: Ensure that the composite package contains the current package and an action history folder.
3) Include Action Patches: Copy the action patches to the action history folder within the composite package.
4) Update Metadata: Modify the `storage.json` and other relevant configuration files to reflect the composite package and its contents.
5) Validate Package: Ensure that the composite package meets all Capsium package requirements and is properly versioned.

Example process: `shell # Collect action patches cp /path/to/action-history/* /composite-package/internal/action-history/

9. Create composite package structure

mkdir /composite-package/internal cp -r /original-package/* /composite-package/internal/

10. Update metadata

jq '.packages += [{"id": "compositeCapsiumPkg1", "internalActionHistoryPath": "internal/action-history"}]' /composite-package/storage.json > /composite-package/storage_tmp.json mv /composite-package/storage_tmp.json /composite-package/storage.json

11. Validate package

capsium-validate /composite-package `

Additional considerations for creating a composite Capsium package:

Dependency Management	Ensure that the composite package correctly references the current package and any other dependencies.
Namespace Handling	Manage namespaces to avoid conflicts between datasets from different packages.
Testing	Thoroughly test the composite package to ensure that the data changes are correctly applied and all functionalities work as expected.

Example of dependency management: `json { "compositePackageId": "compositeCapsiumPkg1", "dependencies": [ { "packageId": "capsiumPkg1", "version": "1.0" } ], "internalActionHistoryPath": "internal/action-history" } `

Example of namespace handling: `json { "datasetNamespaces": { "capsiumPkg1": "namespace1", "compositeCapsiumPkg1": "namespace2" } } `

Example of testing script: `shell # Test script for composite package capsium-test /composite-package