CalConnect TC

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.

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

There are no normative references in this document.

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
}
```

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

NOTE 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 3

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

1) Example 2:


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

Figure 4

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

1) Example 3:

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

Figure 5

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

1) Example 4:

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

Figure 6

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 7

`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 8

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.

Here is an example:

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

Figure 9

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 10

- 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 11

— 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 12

— 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 13

— 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 14

— 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 15

— 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 16

— 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 17

— 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 18

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

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

Figure 19

— 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 20

— 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 21

— 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 22

— 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 23

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

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


:2024

}

Figure 24

5.3.6. Read-only

Capsium packages can be configured as immutable, ensuring that their content cannot be modified after creation.

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 25

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.

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 26

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:

```
{
    "dependencies": {
        "capsium-core": "1.0.0"
    },
    "resources": [
        {
            "path": "node_modules/capsium-core/scripts/main.js",
            "visibility": "exported"
        }
    ]
}
```

Figure 27

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 28

— 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 29

— 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 30

— 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 31

— 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 32

— This specifies that the root file for the package is `index.html` located in the root directory of the package.

Example:

```html
<!DOCTYPE html>
<html lang="en">
```

```html
<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 33

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 34

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 35

```
<!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 36

CSS Files

These files define the styles for HTML elements, specifying colors, fonts, layouts, and other visual aspects.

``Example (`styles.css`):``

Figure 37

```
body {
    font-family: Arial, sans-serif;
    background-color: =f0f0f0;
}
h1 {
    color: =333;
}
```

Figure 38

JavaScript Files

These files contain client-side scripts that add interactivity and dynamic behavior to the web pages.

``Example (`app.js`):``

Figure 39

```
document.addEventListener('DOMContentLoaded', () => {
    console.log('Page loaded');
});
```

Figure 40

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 41

```
example-capsium-package/
├── images/
│   ├── logo.png
│   └── banner.jpg
│
```

Figure 42

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 (`dynamic-content.js`):

Figure 43

```
fetch('https://api.example.com/data')
    .then(response => response.json())
    .then(data => {
        document.getElementById('dynamic-
content').innerText = data.message;
    });
```

Figure 44

Dynamic References

Links and scripts that point to external resources or APIs that provide dynamic data.

Example (`index.html`):

Figure 45

```
<div id="dynamic-content"></div>
<script src="dynamic-content.js"></script>
```

Figure 46

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 (`index.html`):

Figure 47

```
<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 48

Content for Different Languages

Providing content in multiple languages and loading the appropriate version based on the user's language preferences.

Example:

Figure 49

```
example-capsium-package/
├── content/
│   ├── en/
│   │   ├── index.html
│   │   └── about.html
│   ├── es/
│   │   ├── index.html
│   │   └── about.html
```

Figure 50

JavaScript to load language-specific content (`language-loader.js`):

Figure 51

```
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 52

Device-Specific Content

Serving different versions of content based on the type of device (e.g., mobile vs. desktop).

Example:

Figure 53

```
<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 54

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 or handlers, including both static data and dynamic HTTP API routes.

The routing file `routes.json` is the single source of truth for routing in the package, simplifying management and ensuring consistency. This setup covers static resource routing, dynamic API endpoints, and the essential configurations for headers and HTTP methods.

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": [
        // Static resource 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"
        },

        // HTTP API 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 55

In this example, the `routes.json` file includes routes for both static resources (like HTML, CSS, images, and data files) and dynamic HTTP API endpoints.

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 56

— 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.
Access control	Configurations to manage who can access the data and under what conditions.
Response format	Specifies the format in which the data will be returned, such as JSON, XML, or CSV.

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 57

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 58

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.

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	Object containing access control settings.

Table 4 — Table 4: 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.

By configuring these attributes, Capsium packages can effectively manage data storage and provide structured access to data sets through defined routes. This ensures data can be securely and efficiently retrieved by authorized users.

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 59

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 60

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:

```json
{
    "Content-Type": "application/json",
    "Cache-Control": "no-cache",
    "Access-Control-Allow-Origin": "*"
}
```

Figure 61

And `headers/postResourceHeaders.json` might look like this:

```json
{
    "Content-Type": "application/json",
    "Access-Control-Allow-Origin": "*"
}
```

Figure 62

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.

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:

```json
{
    "routes": [
        {
            "path": "/api/public",
            "handler": "publicHandler",
            "visibility": "exported"
```

```
        },
        {
            "path": "/api/private",
            "handler": "privateHandler",
            "visibility": "private"
        }
    ]
}
```

Figure 63

5.7.7. Route Inheritance and Processing

The Capsium package supports inheriting routes from dependency packages, allowing for remapping, rewriting responses, enhancing response headers, or supplanting request headers. This flexibility enables packages to extend and modify the behavior of routes defined in their dependencies.

Requirements and Specifications

1) Inheriting Routes:
   - Inherited routes must be declared in the `routes.json` file of the dependent package.
   - An inherited route should specify the package and the original route path.
2) Remapping Routes:
   - Inherited routes can be remapped to a new path in the dependent package.
3) Rewriting Responses:
   - The system should allow for rewriting the response of an inherited route, either by modifying the content or altering the headers.
4) Enhancing Response Headers:
   - Additional headers can be added to the response of an inherited route to enhance security, performance, or other attributes.
5) Supplanting Request Headers:
   - Request headers can be modified or added before forwarding the request to the inherited route.
6) Example Configuration:
   - An example `routes.json` file demonstrating route inheritance and processing:

```
{
    "dependencies": {
        "capsium-core": "1.0.0"
    },
    "routes": [
        {
            "path": "/api/exported",
            "handler": "node_modules/capsium-core/handlers/publicHandler",
            "visibility": "exported"
        },
        {
            "path": "/api/remapped",
            "handler": "node_modules/capsium-core/handlers/publicHandler",
            "visibility": "private",
            "remap": "/api/newpath"
        },
        {
            "path": "/api/rewritten",
            "handler": "node_modules/capsium-core/handlers/publicHandler",
```



```
            "visibility": "private",
            "responseRewrite": {
                "body": "Modified response content",
                "headers": {
                    "X-Custom-Header": "CustomValue"
                }
            }
        },
        {
            "path": "/api/enhanced",
            "handler": "node_modules/capsium-core/handlers/publicHandler",
            "visibility": "exported",
            "responseHeaders": {
                "Cache-Control": "no-cache",
                "X-Enhanced-Header": "EnhancedValue"
            }
        },
        {
            "path": "/api/supplanted",
            "handler": "node_modules/capsium-core/handlers/publicHandler",
            "visibility": "private",
            "requestHeaders": {
                "Authorization": "Bearer new-token",
                "X-Forwarded-For": "client-ip"
            }
        }
    ]
}
```

Figure 64

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.

5.8. Storage

The Capsium package includes a comprehensive storage system that supports layered storage with a unified merged filesystem view, static data files, and databases. 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.
Static File with Structured Data	Formats such as CSV, JSON, YAML.
Backed by Schemas	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. Layered storage

Layered storage in the Capsium package allows for multiple storage layers to be combined 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.

Requirements and Specifications
: 1) Layer Structure:
  — The storage system must support multiple layers, where each layer can be read-only or writable.
  — Layers should be stacked in a specified order, with the topmost layer being the most recent.
  2) Unified View:
  — The system must present a single filesystem view that merges all layers.
  — File access should respect the layer order, with the topmost layer taking precedence in case of conflicts.
  3) Non-destructive Changes:
  — Changes should be written to the topmost writable layer, preserving lower layers intact.
  — Deletions in higher layers should not physically remove the files from lower layers but should mark them as deleted in the unified view.
  4) Configuration:
  — The configuration for layered storage should be specified in a `manifest.json` file.
  — Example:

```
{
    "layers": [
        {
            "path": "base",
            "writable": false,
```



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

Figure 66

— The `visibility` field determines if the layer is exposed as an inheritable interface (`exported`) or kept private (`private`).

5.8.3. 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 67

1) Configuration:
   - — The location of static files should be defined in the `routes.json` file.
   - — Example:

```
{
    "static": {
        "path": "static",
        "url": "/static"
    }
}
```

Figure 68

5.8.4. 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.

5.8.4.1. 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:

```
{
  "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"
    }
  ]
}
```

Figure 69

Here's an example of a JSON schema for validating a dataset of user information:

```
{
  "$schema": "link:++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"]
```

```
}
```

Figure 70

Here’s an example of a YAML schema for validating a dataset of user information:

```yaml
%YAML 1.2
---
$schema: "link:++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
```

Figure 71

5.8.4.2. 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"
    }
  ]
}
```

Figure 72

Here's an example of an SQL schema for a table of user information:

```
CREATE TABLE users (
  id TEXT PRIMARY KEY,
  name TEXT NOT NULL,
  email TEXT NOT NULL UNIQUE
);
```

Figure 73

Here's an example of a JSON schema for validating the structure of a table of user information:

```
{
  "$schema": "link:++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"]
}
```

Figure 74

5.8.4.3. 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.

5.8.5. 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`.

5.8.5.1. 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:

```
{
  "datasetId": "dataset1"
}
```

Figure 75

5.8.5.2. 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:

```
{
  "datasetId": "dataset1",
  "dataId": "dataItem42"
}
```

Figure 76

5.8.5.3. 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:

```
{
```



```
  "datasetId": "dataset1",
  "data": {
    "key1": "value1",
    "key2": 123
  }
}
```

Figure 77

5.8.5.4. 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:

```
{
  "datasetId": "dataset1",
  "dataId": "dataItem42",
  "data": {
    "key1": "newValue",
    "key2": 456
  }
}
```

Figure 78

5.8.5.5. 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:

```
{
  "datasetId": "dataset1",
  "dataId": "dataItem42"
}
```

Figure 79

5.8.6. Data Persistence

5.8.6.1. 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.

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. 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.

5.8.6.2. 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:

```
{
  "timestamp": "2024-05-28T12:34:56Z",
  "user": "user123",
  "action": "update",
  "datasetId": "dataset1",
  "dataId": "dataItem42",
  "changes": {
    "key1": "newValue"
  }
}
```

Figure 80

5.8.6.3. 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
└── ...
```

Figure 81

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"
  }
}
```

Figure 82

5.8.6.4. 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:

```
{
  "originalPackageId": "capsiumPkg1",
  "newPackageId": "capsiumPkg2",
  "actionHistoryLocation": "/path/to/action-history"
}
```

Figure 83

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:

```
# Collect action patches
cp /path/to/action-history/* /new-package/action-history/

# Create new package structure
mkdir /new-package
cp -r /original-package/* /new-package/

# 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

# Validate package
capsium-validate /new-package
```

Figure 84

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.

:2024

Example of integrity check:

```
capsium-check-integrity /new-package
```

Figure 85

Example of updating documentation:

```
# Capsium Package Documentation

## 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
```

Figure 86

5.8.6.5. 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:

```
{
  "originalPackageId": "capsiumPkg1",
  "compositePackageId": "compositeCapsiumPkg1",
  "internalActionHistoryPath": "internal/action-history"
}
```

Figure 87

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:

```
# Collect action patches
```



```
cp /path/to/action-history/* /composite-package/internal/action-history/

# Create composite package structure
mkdir /composite-package/internal
cp -r /original-package/* /composite-package/internal/

# 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

# Validate package
capsium-validate /composite-package
```

Figure 88

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:

```
{
  "compositePackageId": "compositeCapsiumPkg1",
  "dependencies": [
    {
      "packageId": "capsiumPkg1",
      "version": "1.0"
    }
  ],
  "internalActionHistoryPath": "internal/action-history"
}
```

Figure 89

Example of namespace handling:

```
{
  "datasetNamespaces": {
    "capsiumPkg1": "namespace1",
    "compositeCapsiumPkg1": "namespace2"
  }
}
```

Figure 90

Example of testing script:

```
# Test script for composite package
capsium-test /composite-package

# Validate composite package
```



```
capsium-validate /composite-package
```

Figure 91

5.9. Security

5.9.1. 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.

Digital signatures	The package can be signed digitally to verify its authenticity and integrity.
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.

Example:

```
{
    "security": {
        "digitalSignatures": {
            "publicKey": "path/to/public.key",
            "signatureFile": "package.sig"
        },
        "integrityChecks": {
            "checksumAlgorithm": "SHA-256",
            "checksums": {
                "index.html": "d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2",
                "styles.css": "e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3"
            }
        },
        "sandboxing": {
            "enabled": true,
            "parameters": {
                "memoryLimit": "512MB",
                "cpuLimit": "2"
            }
        }
    }
}
```

Figure 92

5.9.2. Digital Signatures

Requirements and Specifications

1) Digital Signature Implementation:

publicKey
: Path to the public key file used for verifying the digital signature.
  - Type: `string`
  - Example: `"path/to/public.key"`

signatureFile
: Path to the signature file that contains the digital signature.
  - Type: `string`

Example	`"package.sig"`

2) Verification Process:
   - The system must use the public key to verify the digital signature.
   - 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

```
{
    "security": {
        "digitalSignatures": {
            "publicKey": "path/to/public.key",
            "signatureFile": "package.sig"
        }
    }
}
```

Figure 93

5.9.3. 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": "d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2",
    "styles.css": "e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3"
}
```

Figure 94

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": "d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2",
                "styles.css": "e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3"
            }
        }
    }
}
```

Figure 95

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 96

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 97

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.

5.9.4. 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

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

Figure 98

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.

5.10. Encrypted information

5.10.1. 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 99

5.10.2. 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 100

5.10.3. 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 101

5.10.4. 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 102

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 103

5.10.5. Detailed Use Cases and Examples

5.10.5.1. 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 104

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.

5.10.5.2. 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 105

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.

5.10.6. 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

5.10.6.1. 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** (Required if `encrypted` is `true`):

- Type: `object`
- Description: Details about the encryption method used.
- Properties:
  - encryptedWith:
    - Type: `string`
    - Enumeration: `"DEK"`, `"OpenPGP"`
    - Description: The method used for encryption.
    - Example: `"DEK"` or `"OpenPGP"`
  - algorithm:
    - Type: `string`
    - Enumeration: `"OCB"`, `"OpenPGP"`
    - Description: The encryption algorithm used.
    - Example: `"OCB"` or `"OpenPGP"`

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 106

5.10.7. Detailed Example

5.10.7.1. 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"
            }
        ]
    }
```


:2024

```
}
```

Figure 107

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 108

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 109

5.11. Validation

5.11.1. 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.

5.11.2. Data validation

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

5.11.3. 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 110

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

5.11.4. 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.

:2024

Example:

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

Figure 111

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.

5.11.5. 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 112

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.

5.11.6. 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.

5.12. Testing

5.12.1. General

This clause specifies the structure and options for the Capsium package testing YAML format. This format is used to define tests for various aspects of Capsium packages, including HTTP routes, file existence, data validation, and configuration testing.

A YAML-based domain-specific language (DSL) for describing tests to be executed against Capsium packages is specified below. The DSL is designed to be programming language-independent, allowing for implementations in various languages.

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 113

5.12.2. Test Types and Options

5.12.3. 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 114

— `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
```


:2024

```
  response_contains: "Welcome"
```

Figure 115

5.12.4. 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 116

— `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 117

5.12.5. 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 118

— `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 119

5.12.6. 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 120

— 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 121

6. 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 122

7. Conformance

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

7.1. Packaging options

7.1.1. General

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

The `packaging.json` file serves as a centralized configuration for managing the compression, digital signature, and encryption aspects 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. Signing occurs on the entire package, and will be stored in a file called signature.sig inside the package. 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

The digital signature is calculated on the combined data of:

— package/metadata.json
— package/signature.json (where the “signature” key is set to an empty string)
— 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

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



```
  "digitalSignature": {
    "algorithm": "RSA-SHA256",
    "keyLength": 2048,
    "certificateType": "X.509",
    "signatureFile": "signature.sig"
  },
  "encryption": {
    "algorithm": "AES-256",
    "mode": "GCM",
    "keyManagement": "secure",
    "fileExtension": ".enc"
  }
}
```

Figure 123

7.1.2. 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.

7.1.2.1. 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.

7.1.2.2. Configuration Details in `packaging.json`

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

Figure 124

7.1.3. Digital Signature Using X.509 or OpenPGP

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

7.1.3.1. 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.	

7.1.3.2. Configuration Details in `packaging.json`

```
{
  "digitalSignature": {
    "algorithm": "RSA-SHA256",
    "keyLength": 2048,
    "certificateType": "X.509",
    "signatureFile": "signature.sig"
  }
}
```

Figure 125

For OpenPGP:

```
{
  "digitalSignature": {
    "algorithm": "RSA-SHA256",
    "keyLength": 2048,
    "certificateType": "OpenPGP",
    "signatureFile": "signature.sig"
  }
}
```

Figure 126

7.1.4. Encryption

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

7.1.4.1. Requirements and Specifications

Encryption Algorithm	AES-256
Mode of Operation	GCM (Galois/Counter Mode) for authenticated encryption
Key Management	Secure distribution and storage of encryption keys
Encrypted File	The encrypted package should have a `.enc` extension.

7.1.4.2. Configuration Details in `packaging.json`

```
{
  "encryption": {
    "algorithm": "AES-256",
    "mode": "GCM",
    "keyManagement": "secure",
    "fileExtension": ".enc"
```



```
  }
}
```

Figure 127

7.1.5. 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.

7.1.5.1. 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:

```json
{
    "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 128

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.

7.1.5.2. 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.

7.2. User authentication

7.2.1. 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 129

7.2.2. 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.

7.2.3. Apache Authentication (passwd)

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

7.2.3.1. 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.

7.2.3.2. 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 `

7.2.4. External Authentication via OAuth

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

7.2.4.1. 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.

7.2.4.2. Configuration Details in `oauth_config.json`

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

Figure 130

7.2.4.3. 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.

7.2.5. Combined Configuration Example

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

`authentication.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": "link:++https://yourapp.com/oauth/callback"++[],
```



```
      "scope": ["email", "profile"],
      "stateSecret": "YOUR_STATE_SECRET"
    }
  }
}
```

Figure 131

general
: Specifies general authentication requirements,

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.

8. Composite Packages

Composite packages in Capsium allow for the bundling of multiple Capsium packages into a single, cohesive unit. This provides an organized and efficient way to manage dependencies, resources, and configurations.

8.1. Structure (Composite Package of Multiple Capsium Packages)

A composite package is a structured collection of multiple Capsium packages, each contributing its own functionality and resources. The structure typically includes: * A manifest file (`manifest.json`) that outlines the composite package's metadata and dependencies. * A `packages` directory containing the individual Capsium packages. * Configuration files for resource routing and storage management.

Example `manifest.json`:

```
{
  "name": "composite-package-example",
  "version": "1.0.0",
  "description": "A composite package consisting of multiple Capsium packages.",
  "packages": [
    "package1",
    "package2",
    "package3"
  ]
}
```

Figure 132

Directory structure:

```
composite-package/
├── manifest.json
├── packages/
│   ├── package1/
│   ├── package2/
```

```
│    └── package3/
├── routes.json
└── storage.json
```

Figure 133

8.2. Specifying Dependencies in Metadata

Dependencies between Capsium packages within a composite package are specified in the `manifest.json` file. This includes defining which packages are required and any specific versions or constraints.

Example `manifest.json` with dependencies:

```
{
  "name": "composite-package-example",
  "version": "1.0.0",
  "description": "A composite package consisting of multiple Capsium packages.",
  "packages": [
    "package1",
    "package2",
    "package3"
  ],
  "dependencies": {
    "package1": ">=1.0.0",
    "package2": "^2.0.0",
    "package3": "3.x"
  }
}
```

Figure 134

8.3. Resource Routing

Resource routing in a composite package involves defining how HTTP routes and data routes are managed and potentially remapped to avoid conflicts and ensure proper integration.

8.3.1. Remapping Included HTTP Routes

When combining multiple packages, HTTP routes may need to be remapped to avoid conflicts or to better organize the API endpoints. This is done in the `routes.json` file.

Example `routes.json` with remapping:

```
{
  "routes": [
    {
      "originalRoute": "/api/v1/package1/",
      "remappedRoute": "/api/v1/composite/package1/"
    },
    {
      "originalRoute": "/api/v1/package2/",
      "remappedRoute": "/api/v1/composite/package2/"
    }
  ]
}
```

Figure 135

8.3.2. Remapping Included Data Routes

Similar to HTTP routes, data routes may also need to be remapped to ensure data sources are correctly referenced and do not conflict.

Example `routes.json` with data route remapping:

```
{
  "dataRoutes": [
    {
      "originalRoute": "/api/v1/data/package1/",
      "remappedRoute": "/api/v1/data/composite/package1/"
    },
    {
      "originalRoute": "/api/v1/data/package2/",
      "remappedRoute": "/api/v1/data/composite/package2/"
    }
  ]
}
```

Figure 136

8.4. Storage

Storage configurations in a composite package involve managing and potentially customizing which layers from included packages are activated or deactivated.

8.4.1. Selecting Inherited Layers to Activate / Deactivate

Inherited layers from individual packages can be selectively activated or deactivated based on the needs of the composite package. This is specified in the `storage.json` file.

Example `storage.json`:

```
{
  "storage": {
    "activeLayers": [
      "package1.layer1",
      "package2.layer2"
    ],
    "inactiveLayers": [
      "package3.layer3"
    ]
  }
}
```

Figure 137

8.5. Security, Digital Signatures, and Integrity Checks

Ensuring the security and integrity of a composite package involves the use of digital signatures and integrity checks. Each package within the composite package should have its own digital signature, and the composite package itself should also be signed.

Digital Signatures
: Each package and the composite package should be signed using a cryptographic key to ensure authenticity.

Integrity Checks	Hashes (e.g., SHA-256) should be used to verify that the package contents have not been tampered with.

Example `manifest.json` with signatures and hashes:

```
{
  "name": "composite-package-example",
  "version": "1.0.0",
  "description": "A composite package consisting of multiple Capsium packages.",
  "packages": [
    {
      "name": "package1",
      "version": "1.0.0",
      "hash": "sha256-abcdef1234567890...",
      "signature": "MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA..."
    },
    {
      "name": "package2",
      "version": "2.1.0",
      "hash": "sha256-12345abcdef67890...",
      "signature": "MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA..."
    }
  ],
  "compositeSignature": "MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA..."
}
```

Figure 138

8.6. User Authentication

User authentication in a composite package ensures that only authorized users can access the resources and data provided by the included packages. This is typically managed via an authentication service that supports various authentication methods such as OAuth, JWT, or API keys.

OAuth	Use OAuth 2.0 for user authentication, allowing users to log in using their existing credentials from an OAuth provider (e.g., Google, Facebook).
JWT	Implement JSON Web Tokens (JWT) for stateless authentication. Tokens are issued upon successful login and are included in subsequent requests to verify user identity.
API Keys	Use API keys for application-level access control. API keys are issued to applications and are included in API requests for authentication.

Example `auth.json` configuration:

```
{
  "authentication": {
    "methods": ["OAuth", "JWT", "APIKey"],
    "OAuth": {
      "provider": "https://oauth.example.com",
      "clientId": "your-client-id",
      "clientSecret": "your-client-secret"
    },
    "JWT": {
      "secret": "your-jwt-secret",
      "issuer": "your-issuer",
      "audience": "your-audience"
    },
```


:2024

```
    "APIKey": {
      "headerName": "X-API-Key",
      "validKeys": ["key1", "key2", "key3"]
    }
  }
}
```

Figure 139

By configuring these options, a composite package can ensure robust security and proper access control across all included packages. This structured approach helps maintain the integrity of the composite package while providing a seamless experience for end-users.

9. 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.

9.1. 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/
```

Figure 140

9.2. Operation Environments

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

9.2.1. 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.

9.2.2. 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:

```
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');
});
```

Figure 141

9.2.3. 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.
```

:2024

```
4. Access your package at `https://<username>.github.io/<repository>/`.
```

Figure 142

9.3. 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).

Example API response for `/introspect/status`:

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

Figure 143

9.4. 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.

Endpoints
- `/package/:packageId/status`: Returns the status of the specified package.
- `/package/:packageId/metadata`: Returns the metadata of the specified package.
- `/package/:packageId/logs`: Returns the logs related to the specified package.

Example API response for `/package/:packageId/metadata`:

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

Figure 144

9.5. 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/:packageId/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 145

9.6. 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`:

```
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();
};
```

Figure 146

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

The reactor supports various user authentication methods to secure access to its resources:

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.

:2024

Example configuration for OAuth in `auth-plugin.js`:

```
const passport = require('passport');
const OAuth2Strategy = require('passport-oauth2').Strategy;

passport.use(new OAuth2Strategy({
  authorizationURL: 'link:++https://example.com/oauth/authorize'++[],
  tokenURL: 'link:++https://example.com/oauth/token'++[],
  clientID: 'your-client-id',
  clientSecret: 'your-client-secret',
  callbackURL: 'link:++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('/');
});
```

Figure 147

9.8. Decrypting User Data

The reactor includes mechanisms for securely decrypting user data, ensuring it remains protected while in transit and at rest.

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`:

```
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;
}
```

Figure 148

9.9. Updating Modifiable Capsium Packages

The reactor allows for the dynamic updating of modifiable Capsium packages, ensuring that packages can be kept up-to-date without downtime.

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.

Example update API endpoint in `update-api.js`:

```
app.post('/update-package/:packageId', (req, res) => {
  const packageId = req.params.packageId;
  const newPackageData = req.body; // Assume package data is sent in the request
body
  try {
    // Logic to update the package
    res.status(200).send({ message: 'Package updated successfully' });
  } catch (error) {
    res.status(500).send({ message: 'Failed to update package', error });
  }
});
```

Figure 149

9.10. 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 for a Capsium package
FROM node:14

WORKDIR /app

COPY . .

RUN npm install
```


:2024

```
CMD ["node", "index.js"]
```

Figure 150

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.

9.11. 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`.

9.11.1. 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 151

9.11.2. 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 152

9.11.3. 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",
```


:2024

```
      "hash": "1234abcd5678efgh9012ijkl3456mnop6789qrst0123uvwx4567yzab8901cdef"
    }
  ]
}
```

Figure 153

9.11.4. 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.

Example response:

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

Figure 154

9.12. 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.

9.12.1. 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.

9.12.2. 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 155

9.12.3. Detailed Specifications

9.12.3.1. 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.

9.12.3.2. 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.

9.12.3.3. 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”).

9.12.3.4. 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.

9.12.4. Usage

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

Example deployment command:

```
capsium deploy example-package@1.0.0 --config deploy.json
```

Figure 156

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

