PERRINN APPLICATION

ARCHITECTURE REFERENCE

PRODUCT VERSION: 1

DOCUMENT VERSION: 1

Copyright © 2016, PERRINN Limited

This document is released under <licence\_type> license, but remains the property of the copyright holder.

Table of Contents

[1. Introduction 4](#_Toc467015820)

[1.1 Application Concepts 4](#_Toc467015821)

[2. Design Principles 5](#_Toc467015822)

[2.1 Never Risk Data 5](#_Toc467015823)

[2.1.1 Design for failure 5](#_Toc467015824)

[2.1.2 Application Impact 5](#_Toc467015825)

[2.2 Design for Scale 5](#_Toc467015826)

[2.2.1 Application Impact 5](#_Toc467015827)

[2.2.2 Server Impact 6](#_Toc467015828)

[2.3 Return a meaningful status 6](#_Toc467015829)

[2.4 Keep Operations Atomic 7](#_Toc467015830)

[2.5 This is a Client-Driven Process 7](#_Toc467015831)

[2.6 Encrypt Everything 7](#_Toc467015832)

[3. Service Architecture 9](#_Toc467015833)

[3.1 Entities 9](#_Toc467015834)

[3.2 Object Store 9](#_Toc467015835)

[3.2.1 Logical Storage 9](#_Toc467015836)

[3.2.2 Storage Naming 9](#_Toc467015837)

[3.2.3 Storage Limits 9](#_Toc467015838)

[3.2.4 Object Storage Resilience 9](#_Toc467015839)

[3.2.5 Object Store Security 10](#_Toc467015840)

[3.2.6 Access Permissions 10](#_Toc467015841)

[3.2.7 Archiving 10](#_Toc467015842)

[3.3 Data Storage 10](#_Toc467015843)

[3.3.1 Database Platform 10](#_Toc467015844)

[3.3.2 Database Schema 10](#_Toc467015845)

[3.3.3 Schema Versioning 11](#_Toc467015846)

[3.3.4 Database Security 11](#_Toc467015847)

[3.3.5 Database Resilience 11](#_Toc467015848)

[3.3.6 Database Backup/Recovery 11](#_Toc467015849)

[3.3.7 Database Archiving 11](#_Toc467015850)

[3.3.8 Schema Deployment 12](#_Toc467015851)

[3.4 Application Server(s) 12](#_Toc467015852)

[3.4.1 Application Server Architecture 12](#_Toc467015853)

[3.4.2 Implementation 12](#_Toc467015854)

[3.4.3 Resilience 13](#_Toc467015855)

[3.4.4 Platform Diagram 13](#_Toc467015856)

[3.4.5 Message Queuing 13](#_Toc467015857)

[3.4.6 Result Caching 13](#_Toc467015858)

[3.4.7 Platform Requirements 14](#_Toc467015859)

[4. API Structure 15](#_Toc467015860)

[4.1 Conventions 15](#_Toc467015861)

# 1. Introduction

This document describes the principles, design and implementation guidelines for the PERRINN application. It is intended as a reference for developers and administrators of the application suite.

## 1.1 Application Concepts

The PERRINN application is a distributed workflow/coordination/collaboration tool for management of large-scale projects which can have a widely distributed team make up.

There are three main component groups for the application suite; data storage, processing and client presentation.

# 2. Design Principles

This section of the document details the underlying design decisions and requirements which need to be taken into consideration. It should be read by developers prior to commencing work so they can be familiar with not only the technical decisions, but business and design drivers behind the project.

## 2.1 Never Risk Data

At no time is the loss or compromise of data an acceptable risk. Because of the distributed nature of the application platform, there are many potential situations where data could be lost or exposed. The following guidelines will assist in design choices:

### 2.1.1 Design for failure

Service components can, and will fail. The solution architecture provides for fail-over and recovery of each component of the system (except for an end-user’s device). This ensures that data at rest is protected from loss resulting from an application or storage failure. If required, the application is capable of being scaled to multiple geographic regions to provide additional resilience as well as performance.

Data in flight must be protected. Any transaction must return a success or failure condition. The calling component must silently retry the operation wherever possible, not requiring user intervention. In the worst-case scenario, the user can be prompted to re-enter the information later, or queue the request. For real-time functions, queuing might not be a viable option, and in these cases, the option is not presented.

If no confirmation of a transaction is received, the data must be considered “in-flight”. Where a reliable queue mechanism is available (such as MQTT or SQS), acknowledgement of the queue submission will dismiss the “in-flight” status.

### 2.1.2 Application Impact

At all times, back-end services will present a consistent API set to the client. Client developers do not need to participate in recover operations, unless a transaction is not confirmed. In this case, the application simply needs to retry the operation.

## 2.2 Design for Scale

The application suite is designed to operate at a large scale, but to manage the operating costs, extensive use is made of scalable compute architectures such as Amazon Elastic Compute (EC2) or Azure App Services. These solutions will automatically add and remove compute resources as workload requires, as well as load balance network activities across multiple compute resources.

### 2.2.1 Application Impact

The application developer does not need to be directly concerned with the back-end infrastructure, other than to only ever use a published endpoint for access**. Under no circumstances should a host name of a back-end service be used directly by the client application**. Rather, a series of public API endpoints (api.perrinnapp.net) are used for access. This abstracts the client application from the scaling and availability functions. All requests are made to the public endpoint.

### 2.2.2 Server Impact

Service nodes must be stateless. No local data storage is provided, and server developers should assume the local filesystem to be ephemeral, or better still, read-only. Data storage is broken into two services, database and object. Objects are images and related file-type objects which cannot readily be stored within a database table. Objects are long-lived, and generally do not change once written. Frequently updated data should wherever possible be stored in a database which is external to the service node.

Commit data and objects as early as practical in their lifetime to minimise risk of losing uncommitted data

#### 2.2.2.1 Configuration

Server developers shall assume that the configuration of a service node will not change throughout the lifespan of that particular node, but that the configuration may in fact differ from node to node at any point in time. As such, storage locations, database connections should be retrieved at initialisation, and may be held in memory throughout the application instance lifetime. Developers must not assume paths or connections, but must retrieve this information from the service nodes application configuration data.

#### 2.2.2.2 Startup and Shutdown

The service nodes will be created and destroyed as required by the hosting environment. Upon receiving a shutdown event, any outstanding transactions must be committed, queued externally or aborted. No transaction[[1]](#footnote-1) is to be left in an unacknowledged state.

## 2.3 Return a meaningful status

If an operation cannot be completed by a service node, the error returned to the client application will allow the application to determine what action to take next. For example, the JSON-formatted return for the failure of a database update could contain:

{

“FinalStateCode”: “1”,

“FinalStateText”: “Failed to update database”,

“ErrorCode”: “50”,

“ErrorText”: “Connection Refused”

}

Application developers should not return cryptic or internal messages to end users. A message such as *“The data could not be saved on the server”* is appropriate.

## 2.4 Keep Operations Atomic

Data is less at risk if the transaction cycle is kept atomic. That is, if a record is to be updated, either all the affected fields are updated in the same transaction, or none are.

## 2.5 This is a Client-Driven Process

While the use of “push” services is a possibility, data operations are driven by the client. The service nodes cannot send unrequested traffic to a client. There are two reasons behind this.

Firstly, the application is intended to be used with a largely mobile client fleet. Network addressing cannot be assumed to be reliable as client devices will roam to different network locations, and will be assigned different network addresses from time to time.

Secondly, if the client application is not expecting unsolicited communications from back-end services, there is a reduced likelihood of spoofing contact.

## 2.6 Encrypt Everything

Unless noted in this document, no data is to be transmitted or stored in an unencrypted form. It is not technically feasible (at the time of writing) to encrypt data which is process memory, so in-memory structures will not be encrypted. The following table indicates the required encryption techniques for each component:

|  |  |  |  |
| --- | --- | --- | --- |
| Component | Data | Encryption Technology | Comments |
| App Client | Credentials | Host-based crypto library, AES512 or similar | This data includes user ID and related logon information, which must be stored on the device to allow it to authenticate. |
| App Client | Persistent Data | Host-based crypto library, AES512 or similar | Card data, images, etc. stored on local media |
| App Client | Server Communication | Use https communications only | All network communications is encrypted |
| Database | All Data | Server-based encryption of table data | This is performed ONLY if the operation is transparent to the server application code. |
| Database | Disk Storage | Host-based transparent encryption at rest | If a snapshot is extracted, it cannot be recovered without the encryption keys. This method is preferable to the Table encryption. |
| Object Store | All Data | Encrypted at rest | Prevents data from being copied out of object storage for use elsewhere. |
| Compute Units | Disc Storage | Encrypted at rest | Disc images or physical drives as appropriate are encrypted below the operating system. |
| Snapshot/Backup data | All | Encrypted at rest | Prevents unauthorised recovery |
| Web Server | All | Use https only |  |
| Management Data | All | Use ssl or similar for management access |  |
| Log Data | All | When transmitting log data to analysis/monitoring systems, data must be encrypted in transit |  |

# 3. Service Architecture

This section details the functional components and roles in the back-end services.

## 3.1 Entities

An entity is a logical identifier for a user, group or organisation. An entity is assigned a GUID (Globally Unique ID) when it is created. This GUID is randomly generated, is statistically unlikely to ever be duplicated, and cannot be changed after creation. The Entity GUID is the key to referencing any entity in the system.

Because the entity GUID is essentially a complex random value, it is not easily predicted and makes the system less prone to credential guessing.

## 3.2 Object Store

The object store is used to store large, long-lived data objects such as images, audio and video. This data is generally “Write Once, Read Many” in nature, and is usually best not stored within a database, because of the performance impacts on the database services caused by large streams of data.

### 3.2.1 Logical Storage

Data is stored in an ordered tree structure based on the object owner. Every object in the application data space is stored based on its owner. A directory is created in the Object Store the first time that an entity requires object storage.

### 3.2.2 Storage Naming

When saved to the Object Store, a GUID is assigned to the object to be saved. The original name is stored in the relevant object database table, but not in the file system of the Object Store itself. This eliminates name collisions which could occur over time.

### 3.2.3 Storage Limits

The architecture does not pose any limits on the number or size of objects which may be stored in the object store. Account management processes (such as membership levels) may assign limits as required, but these are implemented in software.

The only limits imposed are those of the underlying filesystem and operating system(s).

### 3.2.4 Object Storage Resilience

Over the lifetime of the application, very large amounts of data will need to be stored in the Object Store, and most of it cannot easily be recreated.

This means that the underlying storage mechanism must ensure that data is replicated and universally presented to the application servers regardless of how and where it is physically stored.

### 3.2.5 Object Store Security

All files in the object store must be accessibly by the application servers regardless of location. The application services are responsible for security and access control. It is reliant on the application service developers to prevent unauthorised access to objects.

All objects in the Object Store are to be encrypted by the underlying system to prevent unauthorised access. Encryption keys will be managed by the application management team.

### 3.2.6 Access Permissions

The permission structure in the Object Store, managed by the application server is modelled on the Unix permissions model. There are two access types, Read (R) and Write (W). These access types are assigned to: Owner (O), Primary Group (G) and Public (P).

Where this differs from the Unix model is that a group (team) can be an entity on its own. Therefore, if the group requests the file be stored in the Object Store, the team GUID is the owner of the object. It will have no primary group.

### 3.2.7 Archiving

As with any service, some users will drop off over time. A process is required to archive this data when it is no longer used. An object may be marked as “archived” at any point, and this allows the underlying storage infrastructure to move this object to near-line storage. If required it will need to be recovered, which is a time-consuming, and potentially costly process.

## 3.3 Data Storage

Because the data being managed by the application suite is structured in nature, a conventional SQL-compliant Database environment is used for record storage.

Client developers do not need to be concerned with the actual table structure as this is abstracted by the application service. This allows changes to be made in the underlying platform with minimal (if any) impact on the client applications.

### 3.3.1 Database Platform

The Application Servers will be coded to use Generic JDBC functionality, which will allow for different database platforms to be used. For the production deployment, MySQL Server is the database platform.

For development purposes, developers can also use H2 database as an in-memory system, which means the solution can be prototyped in a single workstation.

### 3.3.2 Database Schema

The current database schema is available from the source code repository.

### 3.3.3 Schema Versioning

Once the application reaches production, the schema will be assigned a version (“n”). No schema change will be implemented that would cause an application relying on schema version n-2 to fail.

Data in rows at version older than n-2 may be moved or archived.

### 3.3.4 Database Security

Database access is only allowed by network rules to computers which are part of:

* The application server cluster
* A database administrator

Access to the data itself is based on user-id/password pairs. The exact method is implementation-specific.

Data stored in the database shall be encrypted in at least one of the two following ways:

* All storage encrypted by the underlying storage system or operating system
* Encryption at a row level in the database.

Encryption at a storage level is preferred since the impact on the system is expected to be lower.

If row-level encryption is implemented, this must be done in such a way as to be transparent to the application services. The only intervention required by the application will be a key exchange at application startup.

### 3.3.5 Database Resilience

The database is critical to the operation of the entire application stack. It will be implemented in such a way which provides fault tolerance and recoverability. No single failure in the database environment is permitted to make the database platform unavailable.

### 3.3.6 Database Backup/Recovery

To enable point-in-time recovery of the database, an online snapshot mechanism must be available. This mechanism will be responsible for placing the database storage into a consistent state prior to making a snapshot.

In production, snapshots will be taken every hour, and retained as per the next table:

|  |  |  |
| --- | --- | --- |
| Frequency | Lifetime | Purpose |
| Hourly | 24 hours | Point-in-time recovery |
| Daily | 1 calendar month | Disaster Recovery / Audit |
| Monthly | 1 calendar year | Disaster Recovery / Audit |

### 3.3.7 Database Archiving

The application servers must provide a mechanism to archive data which is not actively used, but may be required at a future event.

### 3.3.8 Schema Deployment

Schema deployment is a part of initial deployment and application upgrade. In the case of an application upgrade, the first application service to start is responsible for the schema upgrade. Only this server can upgrade the schema, and the process must only be allowed to execute once only.

## 3.4 Application Server(s)

The Application Server implements the business logic of the application platform. The client application does not, and never will interact directly with the database or Object Store. The client application makes a request to the Application Server (via an appropriate http GET or POST to api.perrinnapp.net). Business logic within the Application Server code processes the data and/or objects and returns result sets to the client application as a JSON-formatted string.

Because of the use of JSON strings, it is mandatory that the streams be encrypted always.

### 3.4.1 Application Server Architecture

When combined with the client application, the architecture forms a traditional MVC (Model/View/Controller) platform. The Controller and Model components reside on the Application Server, while the View rendering is performed by the client application.

### 3.4.2 Implementation

The Model and Controller components are written in Java, with the intention of being hosted on a Tomcat application server. This server may be embedded into the java package (such as Spring Framework), or external on a compute instance.

The advantage of this method is that it allows a complete server environment to be hosted on a developer workstation, especially when combined with the H2 in-memory[[2]](#footnote-2) database engine for non-production.

For production purposes, the application servers will be hosted in an auto-scaling, load balanced environment. Databases will also be fully redundant in the production platform.

Being based on a Java platform, the Application Server can execute on a variety of computer platforms and operating systems with little, if any modification. The production implementation will be deployed to a Linux environment, derived from the Fedora or CentOS distributions. This decision is intended to mitigate ongoing license costs.

### 3.4.3 Resilience

Because the Application Servers are stateless (no local data), there is no special need to resilience in the platform. All units of work are dispatched via message queue, and can be processed by the next available Application Server node.

### 3.4.4 Platform Diagram

The logical organisation of the Application Server is shown in the next diagram:

Load Balancer

Application Server

Application Server

Application Server

Memory Caching

Cluster

Database

Database

### 3.4.5 Message Queuing

Because Application Servers may be created or destroyed at any time, there must be a means of persisting incomplete units of work. This is done via a message queue.

Any work which an Application Server cannot complete immediately is placed on the queue for processing by an available Application Server. This ensures that work is no lost if an instance is scaled back and destroyed.

### 3.4.6 Result Caching

With distributed Application Servers and the potential for high frequency updates, database workload for common object queries can be reduced by using a distributed cache mechanism. The cache platform to be used is Redis (http://redis.io). Query results such as geographic lookups and Team ID can be cached to the Redis host(s). Application Servers will query the cache first, and only search the database if no result is found. Upon completing a database query, the result is also posted to the Redis cache.

### 3.4.7 Platform Requirements

|  |  |  |
| --- | --- | --- |
| Component | Application and Version | Comments |
| Production Operating System | CentOS Linux or similar | Scripting work for deployment makes assumptions around system layout. |
| Development Operating System | Any capable of hosting Tomcat HTTP Server | For non-prod systems, the choice is left to the developer. But SIT is performed on a system matching production |
| Primary Language | Java 8 |  |
| HTTP Server | Apache Tomcat 8 |  |
| Production Database | MySQL 5.7 |  |
| Development Database | H2 | May be configured for in-memory |
| Caching | Redis 3.2 | Can be run locally or remotely |
| Production Endpoint | https://api.perrinnapp.net |  |
| SIT Endpoint | https://api.perrinapp.net | This domain was accidentally ordered, so it might as well be used! |
| Development Endpoint | Determined by developer | Most likely will be localhost |
| Base Logging | Apache Commons Logging |  |
| Build Platform | Apache Maven |  |

# 4. API Structure

## 4.1 Conventions

All API calls are made through http(s) requests. The Application Server code contains a Controller module for each API endpoint. All data is passed into the API through an http POST command.

1. This is for normal operating conditions. In the event of a total and unannounced loss of a service node, the application will need to re-submit the transaction because it will have timed out without acknowledgement. [↑](#footnote-ref-1)
2. The main difficulty with an in-memory solution is that it must be pre-populated from within the same JVM. In addition, a database instance cannot be shared across multiple application servers, so there is no consistency. [↑](#footnote-ref-2)