Radiance Specification

Contents

1 Preamble

This specification fulfills two purposes. It is both an informal document to give users an idea behind the reasoning in the decisions and to make comprehension of the architecture of the project easier, as well as a strict specification on a per-function basis that can be used as a documentation and implementation reference. Each chapter includes sections for more technical explanations of the parts, as well as a section for the definitions of all the functions associated with the part.

1.1 Project Aim

Radiance aims to be many things and covers a lot of areas, mostly due to history. At the most stripped down level it is a library that provides an encapsulation and interaction layer with some special parts related to webservices. Building on top of this framework library it includes a standard definition for a framework in itself, while keeping the actual implementation open to the user.

Further, Radiance as a project provides basic implementations of all defined standard interfaces. This should give a good base if you want to use Radiance as a framework, rather than a framework library and are content with the abilities it provides out of the box. One of the major focuses of the project is on proper encapsulation, which should allow the user to change or replace any parts of the framework without having other modules breaking. This was mainly put in place to allow the user of the finished web application to choose between databases, authentication mechanisms, servers and so on.

Finally, the project even offers actual web applications or content management systems (CMS) that should be instantly ready to be deployed onto an active system. Thanks to the other qualities of Radiance, its strength lies in offering many different kinds of CMSs at the same time, making it easily possible to run a forum, blog, gallery and other similar parts on a single instance. Due to the sharing of resources, this also reduces the necessary effort and confusion on part of the end user of the website, as they only need a single account for everything and have a shared environment to administer everything.

In summary, Radiance's aim is to provide and support the full range of web development, from the most basic interaction library up to fully-fledged applications and systems.

1.2 History

Version five of the TyNET[1] framework was dubbed Radiance, in line with the "light" naming scheme. It is the first version to be written in Common Lisp, previous versions being in PHP.

Initially the idea for TyNET was born out of the frustration of not being able to combine different content management systems into a single entity to reduce the work of having to register multiple accounts. Seeing as systems like Wordpress or Joomla only provided less than stellar alternatives, it was simply decided to hack something together quickly.

From there on out TyNET was rewritten from scratch three times, leading up to version four, which included a blog, gallery, wiki, imageboard, forum, markup system, unified comments system, user profiles and more. Each version took about a year to build, usually starting development rather soon after release of the predecessor. Each time adding more and more components.

Each version before this was never in any part specified and every component had more or less intimate knowledge of other parts. As such it was a very finicky process to change things and thinking about deploying it to other servers with different configurations was scary. This effect became less and less of a problem with each version, but it was never deemed good enough for public release.

In order to change this and to provide a system that actually might be useful to the community, everything was scratched once more, this time with the goal of providing a framework that could be easily deployed, extended and adapted.

[1] TyNET being the short form of TymoonNET, named for being the NETwork component of the Tymoon webcomic it was initially made for. This webcomic does not exist anymore.

2 Framework

The Radiance framework is the core component that keeps everything tied together. It should be seen as a library that provides an interaction layer for modules that are made by third-parties and don't necessarily belong to Radiance itself. In its essence it deals with providing a form to encapsulate functionality into modules and to allow for a generic interaction layer between them.

Interaction between modules happens through the interface and triggers system, each providing one way of interaction: Interfaces provide functions to be called on demand from a module out and triggers allow modules to hook into existing processes to add functionality or adapt data according to certain actions.

Being a web framework, Radiance also handles the primary dispatch and analyzing of web calls. While the web server itself is a module loaded in through the interface system, further delegation of the call happens through the framework. For this purpose, Radiance also specifies a system for dealing with requests, URLs and dispatch.

This section specifies the functionality grouped under the name of the radiance framework library. The specification of the necessary interfaces to complete it to a full framework follows in Standard Interfaces.

2.1 Modules

To offer a way to allow extension of the framework itself as well as web applications in a unified way, Radiance uses a system of modules that encapsulates functionality. Each module is defined in the context of a package, an ASDF system and a unique identifier. Each of these components is linked to each other and should allow for identification, loading and extension of a module. In particular the unique identifier is of importance to the interface system and its dispatch mechanism.

By default, all the module code needs to reside in the same package. If you need to split up your module into multiple packages for one reason or another, you have to call `BIND-TO-MODULE`. This sets up the link from the package to the module identifier, which is necessary for a couple of framework macros. Do note that when using multiple packages, resolving with `MODULE-PACKAGE` will always only refer to the package defined by your system.

2.1.1 ASDF

In order to make linking into the radiance framework as simple as possible, an ASDF extension is deployed. This was seen as the best solution since any kind of project will most likely be using ASDF anyway, so by hooking into it we can automatically perform the necessary actions to prepare an external system to fit into the framework. All a module writer has to do is add `:DEFSYSTEM-DEPENDS-ON (:radiance) :class :radiance-module` to their ASDF system definition.

Of course this has a few side effects, since the framework tries to autodetect package and module identifier. If you want to have separate packages for your ASDF system and your main module, you can also manually set the `:MODULE-PACKAGE` option in your system definition. Similarly, the module identifier can be chosen manually with `:MODULE-IDENTIFIER`. By default it is set to the name of the ASDF system.

If you want to completely avoid automatic setup and ASDF radiance-module integration, you can manually establish a module context with `DEFINE-MODULE`.

2.1.2 Module Identifiers

2.2 Interfaces

2.2.1 Definition

2.2.2 Interface Extensions

2.2.3 Component Expanders

2.2.4 Method Definition

2.2.5 ASDF Loading

2.3 Triggers

2.3.1 Namespaces

2.3.2 Hooks

2.3.3 Triggers

2.4 URI

2.4.1 Properties

2.4.2 Matching

2.5 Request Continuations

2.5.1 Purpose

2.5.2 Invocation

2.5.3 Extent of Use

2.6 Server

2.6.1 Interface

2.6.2 Management

2.6.3 Request Handling

2.7 Library

2.7.1 Modules

Class `RADIANCE-MODULE`

Macro `DEFINE-MODULE`

Generic Function `MODULE-NAME`

Generic Function `MODULE-PACKAGE`

Generic Function `MODULE-IDENTIFIER`

Generic Function `MODULE-SYSTEM`

Macro `CONTEXT-MODULE-IDENTIFIER`

2.7.2 Interfaces

Macro `DEFINE-INTERFACE`

Macro `DEFINE-INTERFACE-EXTENSION`

Macro `DEFINE-INTERFACE-METHOD`

Macro `DEFINE-INTERFACE-COMPONENT-EXPANDER`

Function `INTERFACE-COMPONENT-EXPANDER`

Function `INTERFACE-COMPONENT-TYPES`

Generic Function `EFFECTIVE-SYSTEM`

Macro `WITH-INTERFACE`

2.7.3 Triggers

Class HOOK-ITEM

Accessor NAME

Accessor ITEM-NAMESPACE

Accessor ITEM-IDENTIFIER

Accessor ITEM-FUNCTION

Accessor ITEM-DESCRIPTION

Function HOOK-EQUAL

Function HOOK-EQUALP

Function NAMESPACE-MAP

Function DEFINE-NAMESPACE

Function ADD-HOOK-ITEM

Function NAMESPACE

Function REMOVE-NAMESPACE

Function HOOKS

Function HOOK-ITEMS

Function TRIGGER

Macro DEFINE-HOOK

Function REMOVE-HOOK

Function CLEAR-HOOK-ITEMS

2.7.4 URI

Class `URI`

Accessor `SUBDOMAIN`

Accessor `DOMAIN`

Accessor `PORT`

Accessor `PATH`

Accessor `REGEX`

Function `URI-MATCHES`

Function `URI-SAME`

Function `URI->URL`

Function `URI->SERVER-URL`

Function `URI->CONTEXT-URL`

Function `MAKE-URI`

2.7.5 Request Continuations

Class `REQUEST-CONTINUATION`

Accessor `ID`

Accessor `NAME`

Accessor `TIMEOUT`

Accessor `REQUEST`

Accessor `CONTINUATION-FUNCTION`

Function `CONTINUATION`

Function `CONTINUATIONS`

Function `MAKE-CONTINUATION`

Function `CLEAN-CONTINUATIONS`

Function `CLEAN-CONTINUATIONS-GLOBALLY`

Macro `WITH-REQUEST-CONTINUATION`

2.7.6 Server

3 Standard Interfaces

3.1 Database

3.1.1 Connection

The central database has a single, continuous, main connection. While it is possible for implementations to provide mechanisms to allow for multiple simultaneous connections, the way this is done is not specified. The database connection must be stable under concurrency conditions; that is to say, database operations from multiple threads must be supported.

Function `db:connect`

Syntax: `(db:connect database-name)`
`database-name` --- A string naming the database.

Initiates the database connection to the specified database name. Potentially additional information such as host, user and so on are required. These are implementation dependant and will have to be specified through the radiance configuration or some other method unique to the implementation. If the connection should be lost at any point, the implementation is required to re-establish it automatically and silently. Thus the connection must be maintained in one form or another until `DB:CONNECT` is called again or `DB:DISCONNECT` is called.

If the connection fails, an error of type `DATABASE-CONNECTION-FAILED` is signalled. If a connection is already open, the previous connection is closed automatically and a warning of type `DATABASE-CONNECTION-ALREADY-OPEN` is signalled.

Function `db:disconnect`

Syntax: `(db:disconnect )`

Terminates the database connection if one is already established.

A warning of type `DATABASE-CONNECTION-NOT-OPEN` is signalled if the connection was already closed.

Function `db:connected-p`

Syntax: `(db:connected-p )` ⇒ boolean

Returns `T` if the database is connected, `NIL` otherwise.

3.1.2 Collections

A database has a set of collections, each of which is identified by a unique, extended alphanumeric (a-z,-,_) name. This name is case-insensitive.

A collection is made up of a structure that describes the collection's data layout and the actual data, which comes in records. The implementation is not required to enforce a collection's structure on the data, but may at any time signal an error if operations were to occur that conflict the structure. A structure is made up of field declarations, each field consisting of a collection-unique, extended alphanumeric name and a type.

An implementation may support to add indexes on certain fields of the structure. Indexing is not required to be supported, but if it is it may make query operations on the affected fields faster.

Each collection has a field called `_id`, which is unique for each record as well as sortable according to sequence of record insertion. More specifically, the `_id` field is sortable in the way that ascending sorting begins with the oldest record first and vice versa. The type used for the field is implementation dependant.

Most databases only support a specific range of limited data types, which can be more or less properly matched up with CL types. If they cannot be properly matched, the implementation is required to transform the value as specified. Any implementation is required to support or allow the structure type declarations below. An implementation may or may not support any number of additional data types.

Database Structure Type `:INTEGER`

Stores an object of the CL type `INTEGER`. This type supports an optional extra argument that specifies the amount of bytes used to store the integer. It defaults to 4 and must range between 1-8. If the implementation doesn't support the specified amount of bytes, the type is expected to be upgraded to the next bigger one that contains the range.

If the database cannot store a value because it exceeds the range, an error of type `DATABASE-INVALID-VALUE` is signalled. It is not guaranteed that this error occurs if a value exceeds the specified range since the implementation may have had to upgrade it.

Database Structure Type `:FLOAT`

Stores an object of the CL type `FLOAT`. The database should always store it as a double-precision floating-point number, according to IEEE. If the precision of the value should exceed that of a double, the value is expected to be truncated.

Database Structure Type `:CHARACTER`

Stores an object of the CL type `CHARACTER`. The stored character when retrieved has to be translated into a character of the exact same `CHAR-CODE`.

Database Structure Type `:VARCHAR`

Stores an object of the CL type `STRING`. This type requires an extra argument which specifies the maximum number of characters stored. It maps from and to the CL type `STRING`. The same character requirements as for `:CHARACTER` apply.

If an attempt is made to store a string that exceeds the specified length, an error of type `DATABASE-INVALID-VALUE` is signalled.

Database Structure Type `:TEXT`

Stores an object of the CL type `STRING`. The string may be of arbitrary length. The same character requirements as for `:CHARACTER` apply.

Function `db:collections`

Syntax: `(db:collections )` ⇒ list

Returns a list of strings, each of which is the name of a collection in the database.

Function `db:create`

Syntax: `(db:create collection structure &key indices if-exists)`

```
collection  ---  A string naming the collection. Only a-z,- and _ are allowed.
structure   ::=  (field*)
field       ::=  (field-name type)
type        ::=  type-name | (type-name parameter)
indices     ---  A list of field-names to be indexed.
if-exists   ::=  :error | :ignore
```

Creates a new collection in the database. If the implementation supports structure, it is defined according to the given `structure` list. Every collection automatically adds an `_ID` field as specified above.

If the `collection` name is not extended alphanumeric, an error of type `DATABASE-INVALID-COLLECTION` is signalled. If an additional `_id` field is declared in the `structure` definition, the name of a field is not extended alphanumeric, or a field's type declaration is invalid or unsupported, an error of type `DATABASE-INVALID-FIELD` is signalled. If `if-exists` is set to `:error` and the table exists already, an error of type `DATABASE-COLLECTION-ALREADY-EXISTS` is signalled.

Function `db:structure`

Syntax: `(db:structure collection)` ⇒ list

```
collection  ---  A string naming the collection.
```

Returns the structure definition of the collection in the format described in `CREATE`. If the implementation does not support structure, NIL is returned.

If the collection does not exist, an error of type `DATABASE-COLLECTION-NOT-FOUND` is signalled.

Function `db:drop`

Syntax: `(db:drop collection)`
`collection` --- A string naming the collection.

Removes the collection completely, including data, structure, indexes and name.

If the collection does not exist, an error of type `DATABASE-COLLECTION-NOT-FOUND` is signalled.

Function `db:empty`

Syntax: `(db:empty collection)`
`collection` --- A string naming the collection.

Removes all records from the collection.

If the collection does not exist, an error of type `DATABASE-COLLECTION-NOT-FOUND` is signalled.

3.1.3 Records

A collection stores data in the form of records, each record being a map of fields to their values. Each record contains an `_ID` value that uniquely identifies the record within its collection.

Function `db:iterate`

Syntax: `(db:iterate collection query function &key fields skip amount sort accumulate)`⇒
`collection` --- A string naming the collection.
`query` --- A query object, usually generated by `DB:QUERY`
`function` --- A function to pass the records to.
`fields` --- A list of field names that should be returned.
`skip` --- The amount of records to skip.
`amount` --- The amount of records to select.
`sort` ::= `(sorting*)`
`sorting` ::= `(field-name sort-order)`
`sort-order` ::= `:ASC | :DESC`
`accumulate` --- Generalized boolean.

Selects records with `fields` from `collection` that match the `query`, sorting them by `sort`, skipping over the first `skip` entries, limiting the set by `amount` and finally calling `function` on each record. If `accumulate` is non-NIL, the values of the function calls are collected into a list and returned.

Each record is passed to the function in the form of a hash-map with the field names as keys. If `fields` is NIL, all fields are included. If `sort` is NIL, the order of the records is not specified and may be completely random.

If the collection does not exist, an error of type `DATABASE-COLLECTION-NOT-FOUND` is signalled.

Function `db:select`

Syntax: `(db:select collection query &key fields skip amount sort)` ⇒ list

`collection`	---	A string naming the collection.
`query`	---	A query object, usually generated by `DB:QUERY`
`fields`	---	A list of field names that should be returned.
`skip`	---	The amount of records to skip.
`amount`	---	The amount of records to select.
`sort`	::=	`(sorting*)`
`sorting`	::=	`(field-name sort-order)`
`sort-order`	::=	`:ASC | :DESC`

Returns the records as a list of hash-maps. See `iterate`.

Function `db:count`

Syntax: `(db:count collection query)` ⇒ integer

`collection`	---	A string naming the collection.
`query`	---	A query object, usually generated by `DB:QUERY`

Returns the number of records in the `collection` that match the `query`.

If the collection does not exist, an error of type `DATABASE-COLLECTION-NOT-FOUND` is signalled.

Function `db:insert`

Syntax: `(db:insert collection data)` ⇒ `_ID`

`collection`	---	A string naming the collection.
`data`	---	A hash-map or an alist of the data for the new record.

Inserts the `data` as a new record into the `collection`. Returns the value of the `_ID` field of the newly generated record.

If the `data` contains an `_ID` field or an inexistent field, an error of type `DATABASE-INVALID-FIELD` is signalled. If the collection does not exist, an error of type `DATABASE-COLLECTION-NOT-FOUND` is signalled.

Function `db:remove`

Syntax: `(db:remove collection query &key skip amount sort)`

`collection`	`---`	A string naming the collection.
`query`	`---`	A query object, usually generated by `DB:QUERY`
`skip`	`---`	The amount of records to skip.
`amount`	`---`	The amount of records to select.
`sort`	`::=`	`(sorting*)`
`sorting`	`::=`	`(field-name sort-order)`
`sort-order`	`::=`	`:ASC | :DESC`

Removes all records from the `collection` that match the `query`, sorted by `sort`, skipping the first `skip` and only removing a maximum of `amount` of records.

If the collection does not exist, an error of type `DATABASE-COLLECTION-NOT-FOUND` is signalled.

Function `db:update`

Syntax: `(db:update collection query data &key skip amount sort)`

`collection`	`---`	A string naming the collection.
`query`	`---`	A query object, usually generated by `DB:QUERY`
`data`	`---`	A hash-map or an alist of fields to update.
`skip`	`---`	The amount of records to skip.
`amount`	`---`	The amount of records to select.
`sort`	`::=`	`(sorting*)`
`sorting`	`::=`	`(field-name sort-order)`
`sort-order`	`::=`	`:ASC | :DESC`

Changes all records from the `collection` that match the `query`, sorted by `sort`, skipping the first `skip` and only changing a maximum of `amount` of records by setting all fields contained in `data` to the values in `data`.

If the `data` contains an `_id` field or an inexistent field, an error of type `DATABASE-INVALID-FIELD` is signalled. If the collection does not exist, an error of type `DATABASE-COLLECTION-NOT-FOUND` is signalled.

3.1.4 Query Construct

The database interface exposes a query macro that is required to properly translate expressions into database queries. A database operation only ever affects those records that match the given query (the query evaluates to true). Instead of a query, the `:ALL` keyword may be used if all records should be affected.

Macro `db:query`

Syntax: `(db:query query-form)` ⇒ compiled query

Compiles a query form into a format suitable for the database.

The query macro will code-walk and inspect the different arguments. Each query-form may expect either further query-forms or arguments. Arguments will always be evaluated at run-time, with the exception of quoted symbols which will be interpreted as the field of a collection (see the `QUOTE` query-form). An argument can either be a form or an atom. Depending on the evaluated type of the argument the database may perform transformations or signal an error if the type is not supported. Any database implementation has to support in the very least the following types: `string`, `character`, `real`

The return value of this macro is completely implementation dependant.

Query Form `:=`

Syntax: `(:= a b)`

Compares tokens `a` and `b` with each other. This comparison should be the same as `cl:=` for numerical values or `cl:string=` for strings. `a` and `b` must be arguments.

Query Form `:!=`

Syntax: `(:!= a b)`

Inequality comparison. This is functionally the same as inverting the `=` operator. `a` and `b` must be arguments.

Query Form `:>`

, `:<`, `:<=`, `:>=`
Syntax: `(:>/:</:<=/:>= a b)`

Numerical comparison, same as their `cl` equivalents. `a` and `b` must be arguments.

Query Form `:MATCHES`

Syntax: `(:MATCHES a b)`

Matches `a` against a regex form `b`. The precise regular expression capabilities depend on the implementation, but basic PCRE should be supported. `a` and `b` must be arguments.

Query Form `:IN`

Syntax: `(:IN a &rest arguments)`

Checks if `a` is `=` to one of the provided `arguments`. `a` and `arguments` must be arguments.

Query Form `:AND`

Syntax: `(:AND &rest query-forms)`

Evaluates to true if every sub-form is true. `query-forms` must be query forms.

Query Form `:OR`

Syntax: `(:OR &rest query-forms)`

Evaluates to true if one of the sub-forms is true. `query-forms` must be query forms.

Query Form `:NOT`

Syntax: `(:NOT query-form)`

Evaluates to true if the sub-form evaluates to false and vice-versa. `query-form` must be a query form.

Query Form `QUOTE, :FIELD`

Syntax: `(QUOTE/:FIELD value)`

Translates to a reference to the collection's field, rather than a literal argument value. The `value` may either be a `symbol` or a `string`; in the case of a symbol the symbol's name is used. The field name is forced to lowercase.

Keyword `:ALL`

Translates into "no query restriction" or simply "all records".

3.2 Data Model

The Data Model is a simple, object-oriented interface for database access. It wraps every record into a data-model instance, which handles all standard record operations. Creating new records through data-models happens over a data-model hull, which is a model without an `_ID`.

Class `dm:data-model`

Base class for data-model instances. Any implementation is required to subclass this to provide their own data-model. Data is accessible through the generic `FIELD`.

Function `dm:id`

Syntax: `(dm:id data-model)⇒ _id`
`data-model` --- A `data-model` instance.

Returns the `_ID` field of the current record associated with the data-model instance if there is any. In the case of a hull that has not yet been inserted, NIL is returned.

Accessor `dm:field`

Syntax: `(dm:field data-model field)` ⇒ value
`data-model` --- A `data-model` instance.
`field` --- A field name as string.

Accessor for the fields of the data-model record. If the record does not contain the requested field, NIL is returned instead. This function does not check for field name validity. SETF-able.

Function `dm:get`

Syntax: `(dm:get collection query &key skip amount sort)` ⇒ list
`collection` --- A string naming the collection.
`query` --- A query object, usually generated by `DB:QUERY`
`skip` --- The amount of records to skip.
`amount` --- The amount of records to select.
`sort` ::= `(sorting*)`
`sorting` ::= `(field-name sort-order)`
`sort-order` ::= `:ASC | :DESC`

This function works the same as `DB:SELECT`, but instead of a list of record hash-tables, it returns a list of data-model instances. See `DB:SELECT` for more info.

Function `dm:get-one`

Syntax: `(dm:get-one collection query &key skip sort)` ⇒ data-model
`collection` --- A string naming the collection.
`query` --- A query object, usually generated by `DB:QUERY`
`skip` --- The amount of records to skip.
`sort` ::= `(sorting*)`
`sorting` ::= `(field-name sort-order)`
`sort-order` ::= `:ASC | :DESC`

Selects the first record and creates a data-model of it. If no record at all is matched, NIL is returned instead. See `DB:SELECT` for more info.

Function `dm:hull`

Syntax: `(dm:hull collection)` ⇒ data-model
`collection` --- A string naming the collection.

Returns a new data-model hull for the given `collection`. No test is made whether the collection actually exists or not. The `_ID` of the hull is set to NIL.

If the `collection` name is not extended alphanumeric, an error of type `DATABASE-INVALID-COLLECTION` is signalled.

Function `dm:hull-p`

Syntax: `(dm:hull-p data-model)` ⇒ boolean
`data-model` --- A `data-model` instance.

Returns T if the data-model is a hull that has not yet been inserted, NIL otherwise.

Function `dm:save`

Syntax: `(dm:save data-model)` ⇒ data-model
`data-model` --- A `data-model` instance.

Saves the record to the database, thus `DB:UPDATE`-ing all its fields to the values stored in the `data-model`. The same data-model is returned.

If the supplied data-model is a hull, an error of type `DATA-MODEL-NOT-INSERTED-YET` is signalled.

Function `dm:delete`

Syntax: `(dm:delete data-model)` ⇒ data-model
`data-model` --- A `data-model` instance.

Deletes the record from the database, thus `DB:REMOVE`-ing it. This operation will set the `_ID` of the data-model to NIL, turning it into a hull. The same data-model is returned.

If the supplied data-model is a hull, an error of type `DATA-MODEL-NOT-INSERTED-YET` is signalled.

Function `dm:insert`

Syntax: `(dm:insert data-model &key clone)` ⇒ data-model
`data-model` --- A `data-model` instance.
`clone` --- A generalized boolean.

Inserts the `data-model` as a new record into the database, thus `DB:INSERT`-ing it. If `clone` is non-NIL, a new copy of the data-model is created with the `_ID` of the new record set; this copy is then returned. If `clone` is NIL, the `_ID` field of the data-model is set to that of the new record and the data-model is returned. The supplied data-model can be a hull or a non-hull.