Web Resource Modeling Language ©

WRML ©

"*Wormle*" ©

by
Mark Massé

```
{
    "id" : "http://www.wrml.org/wrml.pdf",
    "version" : 3
}
```

All concepts are from www.WRML.org

Hello WRML

WRML, pronounced like "Wormle", is an **open source software project** focused on providing standards, frameworks, and tools that support the development of web-oriented, client-server applications.

WRML is a schema-based modeling language that comes with a set of standards, tools, and frameworks. WRML can be thought of as a "**Domain Specific Language**" (DSL) for the Web and it's architectural style (known as REST). WRML shares some traits with traditional "Object-Relational Mapping" (ORM) frameworks; but WRML skews closer to Web-Oriented concepts (Schemas in place of Classes or Tables) and elevates the base class "Object" to a schematically-aligned "Model" that was designed with MVC in mind.

WRML.org is the home of the WRML Project, an open source endeavor promoting the development of WRML-based standards, tools, and frameworks.

You're Invited!

What : WRML Project

Where : http://www.wrml.org

RSVP : mark@wrml.org

WRML aims to help establish a uniform, programmatic interface for the Web, or at least a pseudo-standard approach to designing uniform REST APIs. With widespread adoption of the WRML standards, we can leverage a shared REST API design methodology and begin to fashion a **uniformly programmable Web**.

Uniform REST API design is not the ultimate goal, it is only a means to an end. The greatest benefit of a standardized design and implementation methodology is the widespread availability of helpful frameworks and tools that **increase developer productivity** by empowering programmers with a rich set of development tools and frameworks, such as the graphical design tools provided by wrml.org, that we can leverage to design and develop REST APIs and their clients.

Goals as a REST API Platform

Provide REST API Authors with…

Z Built-in, consistent adherence to “RESTful” conventions (URI, HTTP, and metadata designs)

Z Resource state representation in variable formats (e.g. JSON or XML)

Z Tagging representational instances for quick comparison (via ETag header)

Z Cacheable documents with configurable TTLs

Z Content-Type (schema & format) negotiation (via Accept header)

Z Conditional responses using a document’s eTag or date-time stamps

Z Performing asynchronous operations on resources

Z Interface version and dependency management

Z Auto-generated API documentation (HTML)

Z Schema-aware partial response options (slicing and dicing a response data model’s fields)

Z Support for aggregation/composition of data models (representational state)…

> Z At schema design time, by composing the desired schemas together into an aggregate/derived schema
>
> Z At request-time, with server-side hypermedia expansion (link traversal) that turns linked documents into embedded field values prior to response serialization
>
> Z At request-time, by POSTing an ad hoc list of URIs to fetch each model in a list/array

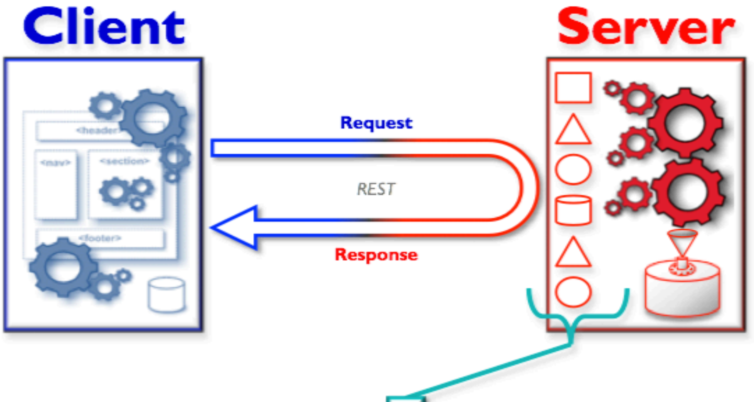
Provide Service Implementors with…

Z A configuration driven REST API platform that handles communication with clients so that the Service can focus on implementing an application’s core logic

Z Instrumentation, metrics collection, and operational dashboards wherever appropriate

Z Thorough documentation with good examples to follow

REST & WRML



Client

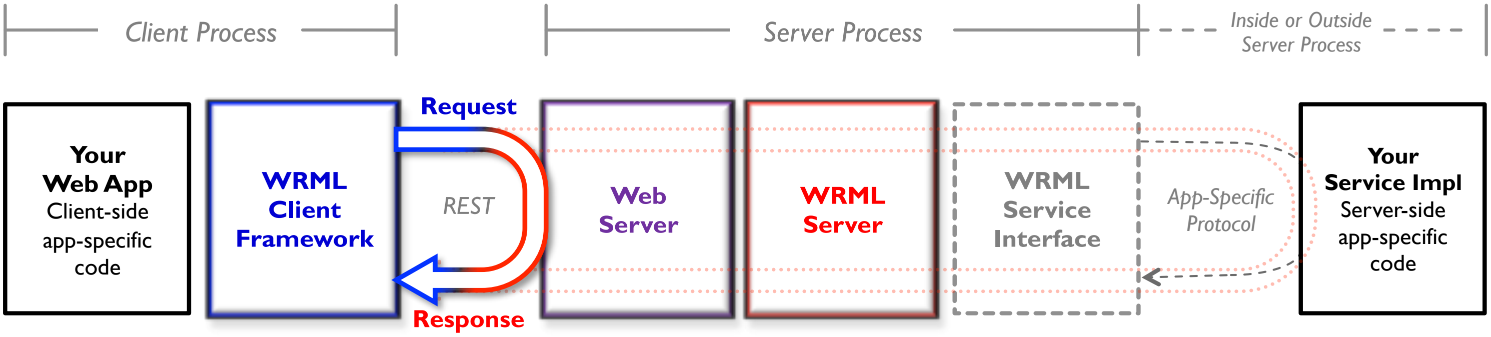
Server

Request

REST

Response

- ☐ **Identifier Design with URIs**
- ☐ **Interaction Design with HTTP**
- ☐ **Metadata Design**
- ☐ **Representation Design**
- ☐ **Client Concerns**



Client Process

Server Process

Inside or Outside Server Process

Your Web App Client-side app-specific code

Request

REST

Response

WRML Client Framework

Web Server

WRML Server

WRML Service Interface

App-Specific Protocol

Your Service Impl Server-side app-specific code

Client-side app logic.

Responsible an app experience that is easy and fun to use.

WRML's uniform **REST** communication framework.

Handles HTTP-based communication and serialization of **Models**.

Delegates incoming requests to the **Server Framework**.

As an example, this could be a simple ***Java servlet***.

Dynamically loads WRML's REST **APIs** and allows reloading of the API's model at runtime.

Delegates all requests to pluggable **Services** based on **Media Type**.

Interface responsible for resolution of resource state (e.g. **Models**).

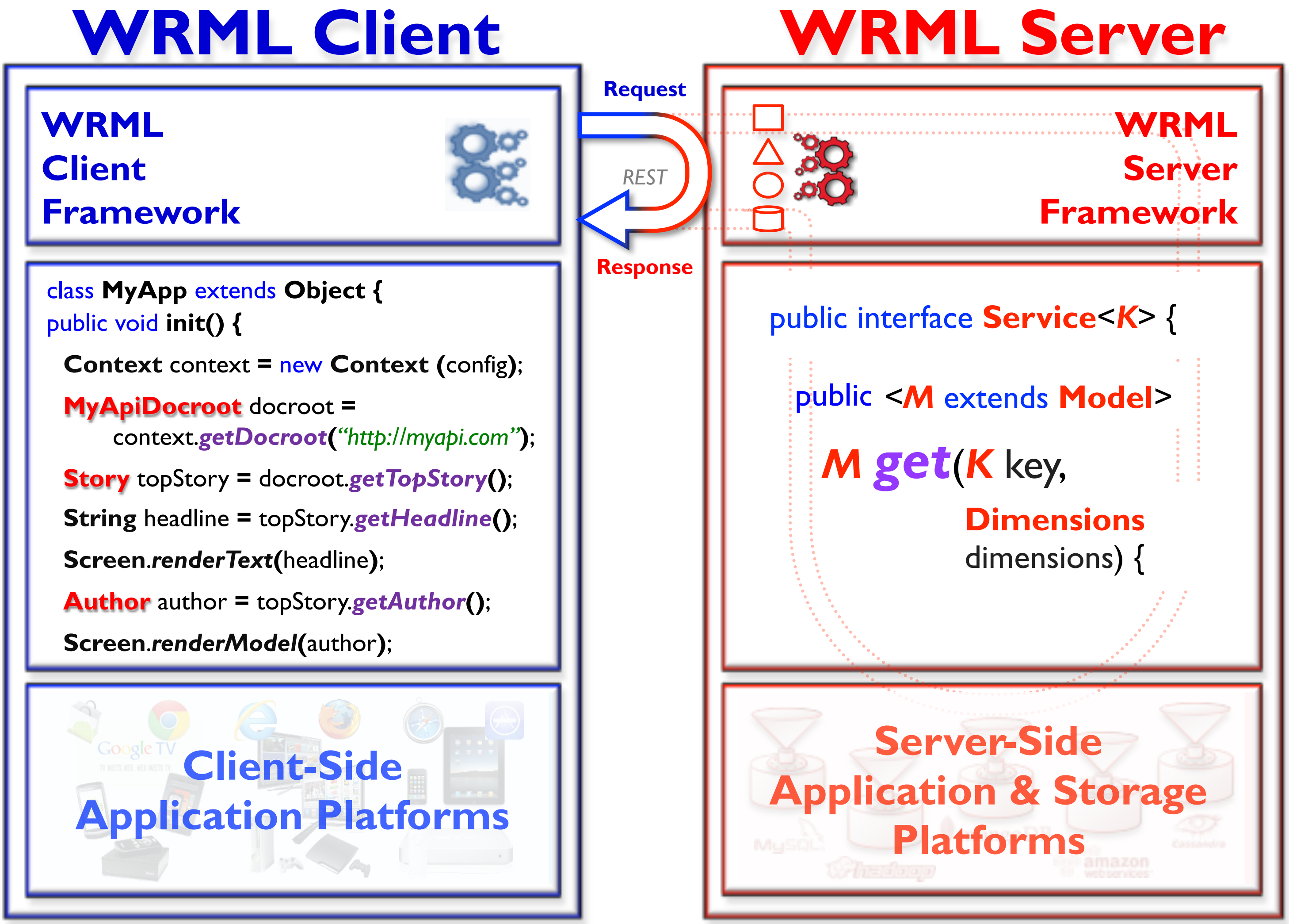
Implementations commonly delegate to an existing server-side codebase to map URIs and **Media Types** to application-specific IDs and data structures.

Server-side app logic.

Responsible for the maintenance of state for a set of exposed Web resources **Models**.

Note that WRML is not required to be used by the clients of servers that leverage WRML to provide REST APIs. A WRML server's non-WRML clients will still find well-formed (no nonsense) JSON or XML (or whatever) with standard media types (e.g. application/json) communicated in the HTTP Content-Type header.

Application Logic & WRML



WRML Client

WRML Client Framework

```
class MyApp extends Object {
public void init() {
  Context context = new Context (config);
  MyApiDocroot docroot =
       context.getDocroot("http://myapi.com");
  Story topStory = docroot.getTopStory();
  String headline = topStory.getHeadline();
  Screen.renderText(headline);
  Author author = topStory.getAuthor();
  Screen.renderModel(author);
```

Client-Side Application Platforms

WRML Server

WRML Server Framework

```
public interface Service<K> {

  public <M extends Model>

    M get(K key,
              Dimensions
              dimensions) {
```

Server-Side Application & Storage Platforms

Client-side, an **app** needs some *app-specific* code to make it do something interesting or unique. Each **platform** (e.g. *iScreen7*) may require some or all of the app/platform specific-code be re-written in order for the app to be as awesome as possible on all platforms.

If a REST API's client does not use a "REST framework", then it's developers work with the Web's "assembly language" and code directly in HTTP. Consistency and REST protocol *correctness* will likely be the first concerns cut. Regardless, somewhere in the app-specific code, you'll likely find code that looks like this example. Through reusable frameworks or with many lines of our own code, it usually ends up here, with MVC.

WRML handles the REST

The application's server needs some app-specific code to provide the app (client) access to some set of "resources" (server-managed data records and *remote* functions).

The WRML server decouples an API's *design* from the back-end **Services**, which are configured to handle certain types of data (schemas of models).

If a REST API's server does not use an API platform/framework like WRML, then it's developers are tasked with writing code to handle some degree of HTTP request routing, content type negotiation, cross-service model aggregation, model slicing/dicing, hypermedia, Collection searching, async requests, etc.

application/wrml

format

The format parameter identifies a **Format** used to serialize a model.

```
{
  "id" : "http://.../application/json",
  "type" : "application",
  "subType" : "json",

  ...Optional links to serialization code
  on demand in languages that support
  "mobile code" (e.g. Java and JavaScript)
  ...
}
```

XML, JSON, and many other useful formats already exists, so rather than introduce a new format, WRML instead uses its media type to establish a pluggable framework for any/all current/future serialization formats.

As a WRML **Document**, the **Format** identifies its media type/subtype, provides links to related resources (e.g. RFCs), and may contain links to downloadable code that clients and/or servers may use to *dynamically learn* to "speak" the format.

The application/wrml *media type* may be used in the **Content-Type** and **Accept** HTTP headers to convey a message body's *syntax* & *structure* independently and simultaneously through its two required parameters:

format: The value is a URI-based identifier of a **Format** document model (owned by the Format REST API).

schema: The value is a URI-based identifier of a **Schema** document model (owned by the Schema REST API).

For example:

application/wrml;
format="http://.../**application/json**;
schema="http://.../**FootballPlayer**"

For parameterized schema's (aka "generic"), the media type supports other *optional* parameters to indicate the schema of each parameterized type using the syntax below:

; *<Type Parameter Identifier>*=<Schema Id>

For example:
; *T*="http://.../**FootballTeam**"

Finally, note that the media type's *schema URI value* may optionally include form-style **query parameters** as outlined below:

{?*embed*}: A comma-separated list of **Link** field names to traverse and embed the response **Document**.

Either {?*include*} or {?*exclude*}: A comma-separated list of field names to **keep** or **discard** for serialization purposes.

schema

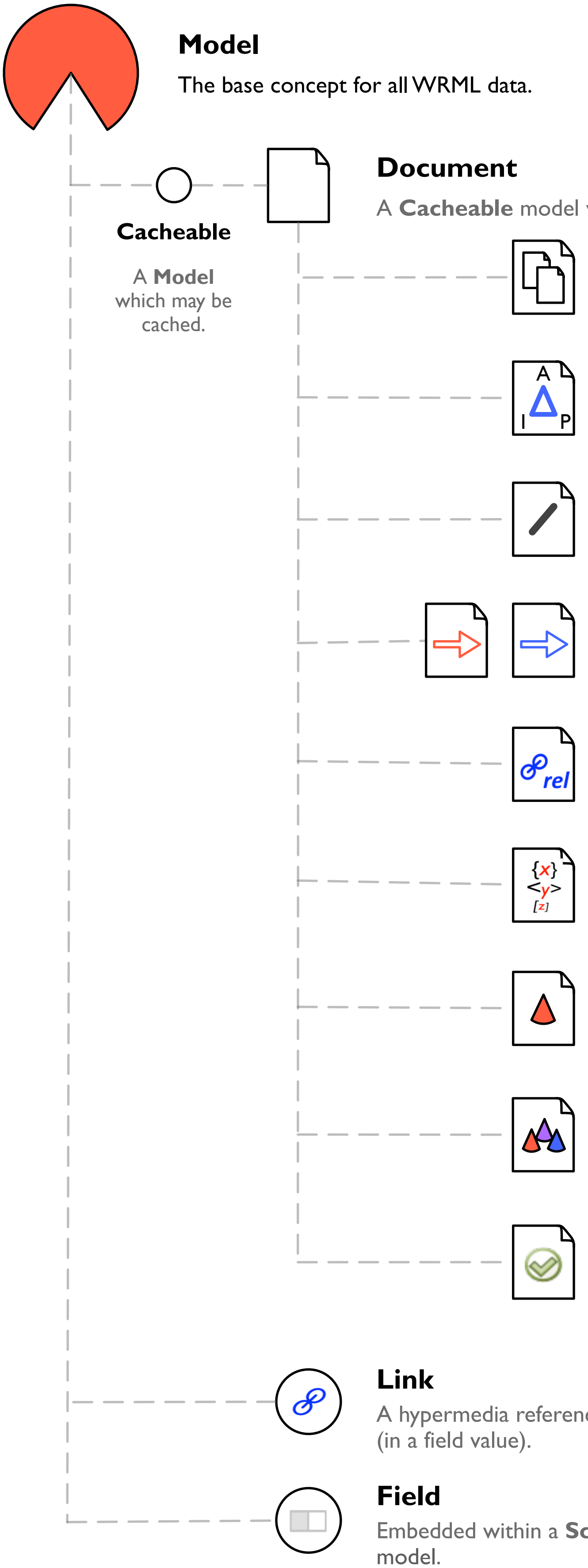
The schema parameter identifies a **Schema** that describes the structure of a model.

```
{
  "id" : "http://.../FootballPlayer",
  "namespace" : "com/.../football",
  "name" : "FootballPlayer",

  ... The schema's Field definitions ...
}
```

XML, JSON, and many other formats already have associated approaches to describing data. Unfortunately most of the approaches tie the data descriptions to the format itself (e.g. XML's DTDs and Schemas, JSON schema, Avro schema, etc.).

As a WRML **Document**, a **Schema** is a description of the data structure of a class of models. Managed as a web resource, **Schemas** may be equivalently represented in a variety of formats, such as XML, JSON, or a Java "POJO" interface (.class, .java, javadoc).

WRML's Core Concepts Icon Glossary



Model

The base concept for all WRML data.

Cacheable

A **Model** which may be cached.

Document

A **Cacheable** model with a URI-based "id" field.

Collection

A **Document** representing a multitude of **Documents**.

Api

A **Document** containing the metadata associated with a REST API's design.

ResourceTemplate

A single "node" in a **Api's** path-based resource hierarchy.

LinkTemplate

Captures a link's **Api** design-time metadata; often regarding the *intra-Api linking* of a "referrer" and an "endpoint" **ResourceTemplate** (via **LinkRelations**).

LinkRelation

A unique, perhaps reusable link specification; like a sharable, web-oriented "function signature".

Format

Describes a serialization (aka "wire") encoding for data.

Schema

Describes the structure of a certain class/category of data.

Namespace (of Schema)

A Collection representation containing some number of **Schema**.

Constraint

Applies some limitation to the **Field** or **Schema** that it is associated with. For example, you might constrain a text field's value to conform to URI syntax or be less than 100 characters long. Constraints are also applied to Link fields to declare a specific **LinkRelation**.

Link

A hypermedia reference (with an identified **LinkRelation**) embedded within a **Document** (in a field value).

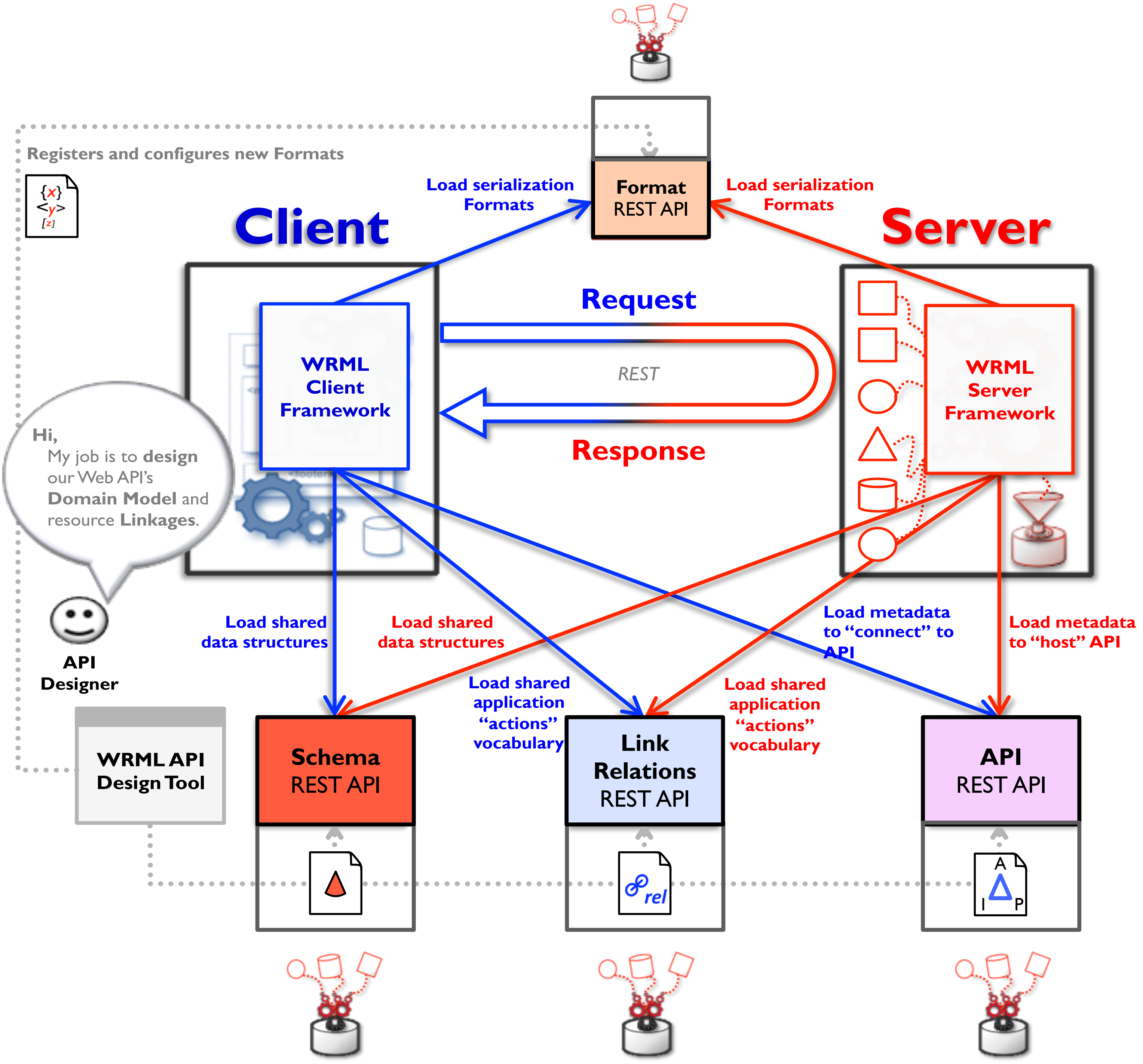
Field

Embedded within a **Schema** to describe a field that may be found in any conforming model.

WRML's REST API-Oriented Architecture

WRML uses WRML to simplify its own REST API-oriented architecture. The diagram below depicts some of the core REST APIs being used by a client and a server that both use the WRML framework.

Note that WRML is not required to be used by the clients of servers that leverage WRML to provide REST APIs. A WRML server's non-WRML clients will still find well-formed (no nonsense) JSON or XML (or whatever) with standard media types (e.g. application/json) communicated in the HTTP Content-Type header.


Registers and configures new Formats
{x}
<y>
[z]
Load serialization Formats
Format REST API
Load serialization Formats
Client
Server
WRML Client Framework
Request
REST
Response
WRML Server Framework
Hi,
My job is to design our Web API's Domain Model and resource Linkages.
API Designer
Load shared data structures
Load shared data structures
Load metadata to "connect" to API
Load metadata to "host" API
Load shared application "actions" vocabulary
Load shared application "actions" vocabulary
WRML API Design Tool
Schema REST API
Link Relations REST API
API REST API
rel
A
I
P

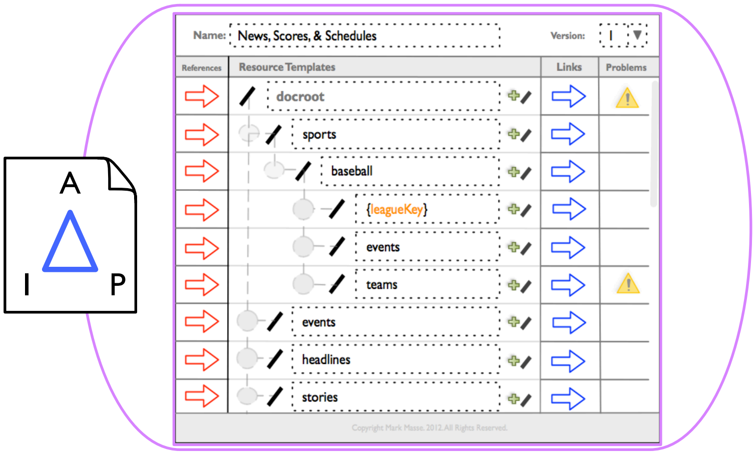
API Design Tool

REST API Design with WRML

Introduction

Why does WRML promote a **UI tool-oriented** REST API design approach in favor of a custom coded solution?

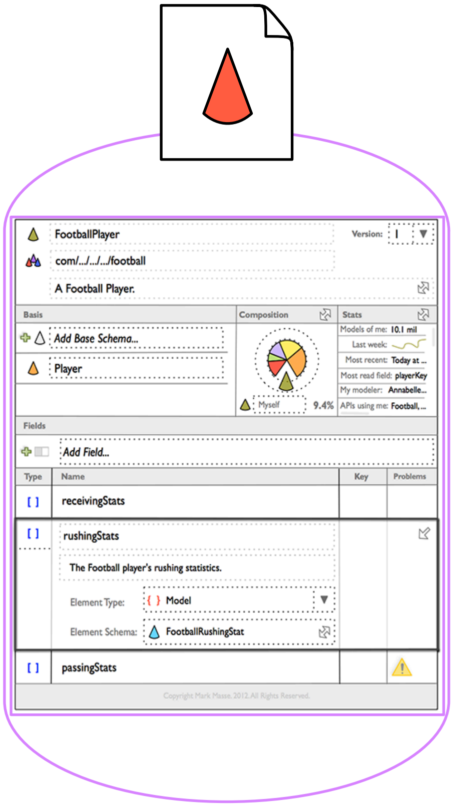
- △ Because a "REST API builder" ***app*** can be easier and more fun to use than IDEs and programming frameworks. As "first-class" constructs in a UI tool, REST APIs may be more easily managed and maintained.
- △ Because specifying a REST API's URI paths along with its input and output data structures shouldn't require server code; it is a *data modeling* task.
- △ Because programmers are skilled at writing code (aka programming) but they are not necessarily skilled at (or interested in) data modeling.
- △ Because our best and brightest data modelers may not be interested in programming.



Name: News, Scores, & Schedules — Version: 1

References	Resource Templates	Links	Problems
⇨	docroot	⇨	⚠
⇨	sports	⇨	
⇨	baseball	⇨	
⇨	{leagueKey}	⇨	
⇨	events	⇨	
⇨	teams	⇨	⚠
⇨	events	⇨	
⇨	headlines	⇨	
⇨	stories	⇨	

Copyright Mark Masse. 2012. All Rights Reserved.



FootballPlayer — Version: 1

com/.../.../.../football

A Football Player.

Basis: Add Base Schema... ; Player

Composition: Myself 9.4%

Stats: Models of me: 10.1 mil; Last week: ; Most recent: Today at ...; Most read field: playerKey; My modeler: Annabelle...; APIs using me: Football, ...

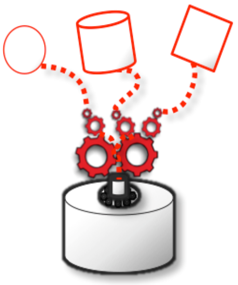
Fields: Add Field...

Type	Name	Key	Problems
[ ]	receivingStats		
[ ]	rushingStats The Football player's rushing statistics. Element Type: { } Model Element Schema: FootballRushingStat		
[ ]	passingStats		⚠

Copyright Mark Masse. 2012. All Rights Reserved.

WRML uses its **Web Resource Server Engine** to *implement* REST API designs

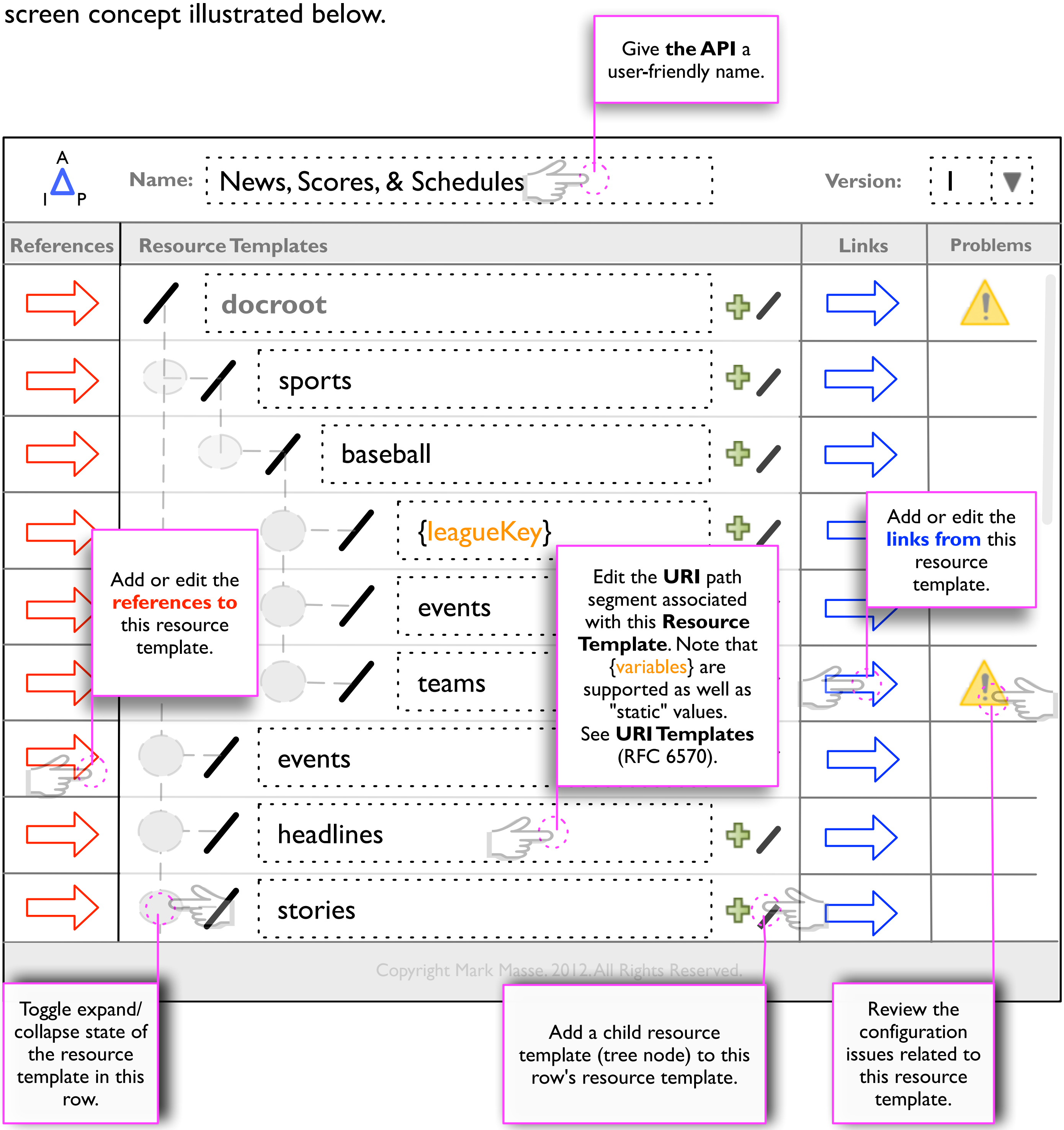
- ○ WRML's server engine is configured at start-up to "host" one or more **Apis** (metadata designs). The engine uses the Api's design information to respond to incoming REST requests.
- ○ WRML's server engine implements a consistent and uniform REST API feature set. As a result of this reuse, any efforts to improve this engine will benefit all WRML servers and their clients.
- ○ WRML's server engine requires no *additional* metadata to be input, beyond what is already needed for the REST API's own self-description.
- ○ WRML's server engine is powered by the *loosely coupled* combination of the Api's metadata loaded in the front-end (as interface) and the **Services** configured in the back-end (as implementation).
- ○ WRML's server engine delegates to configured Services to resolve the inbound CRUD + invoke data model-related requests. By emulating the web's uniform interface, the Service interface enables implementation flexibility (e.g. pluggable back-end storage).



REST API Design with WRML

API Binder

"API Binder" is the codename of the WRML-based tool that provides the API design screen concept illustrated below.



Name: News, Scores, & Schedules

Version: 1

Give **the API** a user-friendly name.

Add or edit the **references to** this resource template.

Edit the **URI** path segment associated with this **Resource Template**. Note that {variables} are supported as well as "static" values. See **URI Templates** (RFC 6570).

Add or edit the **links from** this resource template.

Toggle expand/collapse state of the resource template in this row.

Add a child resource template (tree node) to this row's resource template.

Review the configuration issues related to this resource template.

API Design Tool

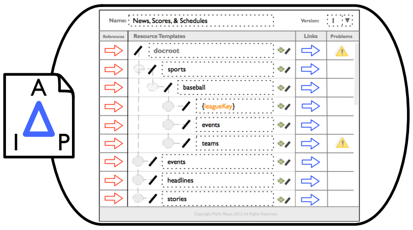
REST API Design with WRML

How?

1

Start

Start the REST API design tool:



Name: News, Scores, & Schedules — Version: 1

References	Resource Templates	Links	Problems
⇨	docroot	⇨	⚠
⇨	sports	⇨	
⇨	baseball	⇨	
⇨	{leagueKey}	⇨	
⇨	events	⇨	
⇨	teams	⇨	⚠
⇨	events	⇨	
⇨	headlines	⇨	
⇨	stories	⇨	

Copyright Mark Masse. 2012. All Rights Reserved.

Search by tags (e.g. "Football") for an existing REST API that may meet your application's needs. The goal here is to facilitate the reuse of APIs.

2

New REST API…

If you didn't find an existing REST API that suits your client app's uses cases, then you will probably want to design a new REST API.

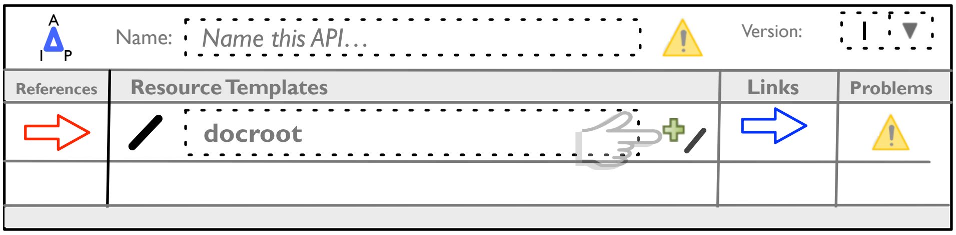
Name this ***API****...*

Name, categorize, and tag your new REST API appropriately.

3

Growing Resources from the Root

All new REST API's begin with a **docroot**; a URI path value of "**/**" (forward slash). This is the URI tree's root path segment.



Name: *Name this API…* — Version: 1

References	Resource Templates	Links	Problems
⇨	docroot	⇨	⚠

You can add URI or URI Template-based path segments (called "resource templates" in WRML) at any level; starting with underneath the API's docroot.

Fill in this ***Resource Template****...*

"Filling in", in this case means simply typing a name (e.g. **stories**) or a URI Template-syntax conforming variable (e.g. **{**storyKey**}**).

For more information regarding URI Templates, see: http://tools.ietf.org/html/rfc6570.

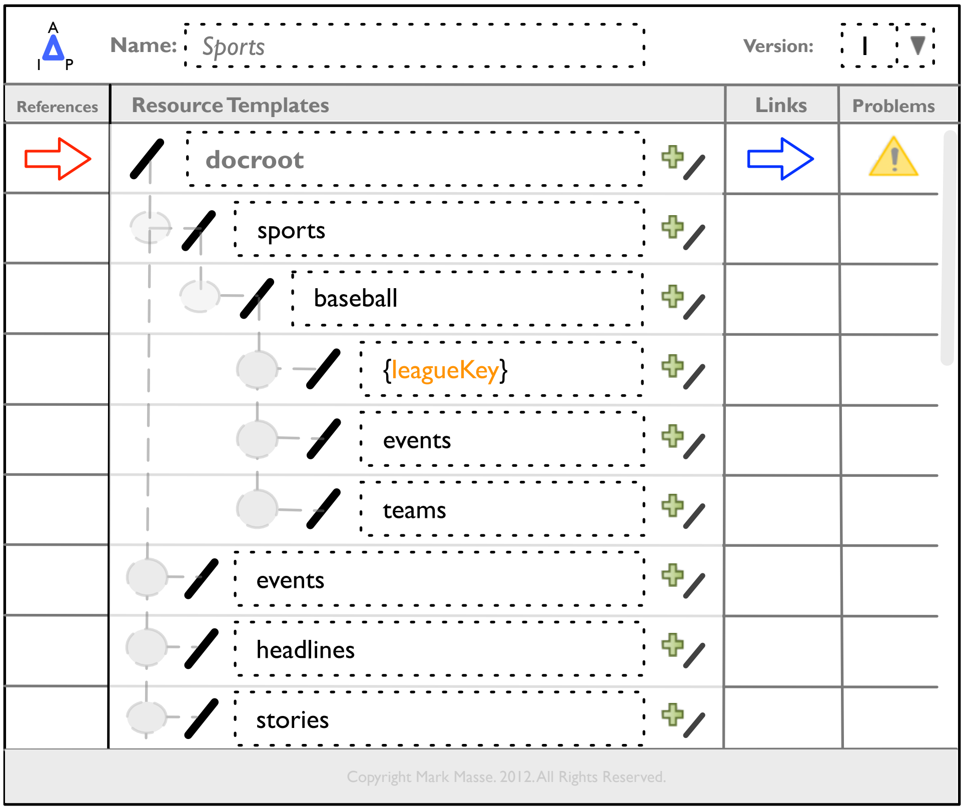
API Design Tool

REST API Design with WRML

How? (continued)

4 *A Resource Template "Tree"*

After some time spent modeling the URI tree's design, you will end up with a set of resource templates, as illustrated below:



Name: Sports

Version: 1

5 *Variable Path Segments*

As noted earlier, a resource template's path segment may include URI template variable syntax (e.g. {leagueKey}).

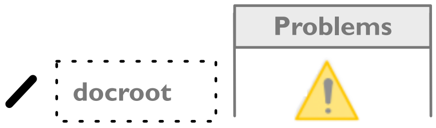
Design Note: In order for WRML's runtime to automate a link *traversal* from a **Document** to a resource identified with a URI template, the variable names must match the names of fields within the referrer Document. To follow links, the WRML runtime substitutes the endpoint's URI Template variables with field values from the referrer (link-referencing) Document. This constraint semantically mirrors an HTML document's need to have the link's entire "href" value available as local state within the representation in order to make *click* work.

REST API Design with WRML

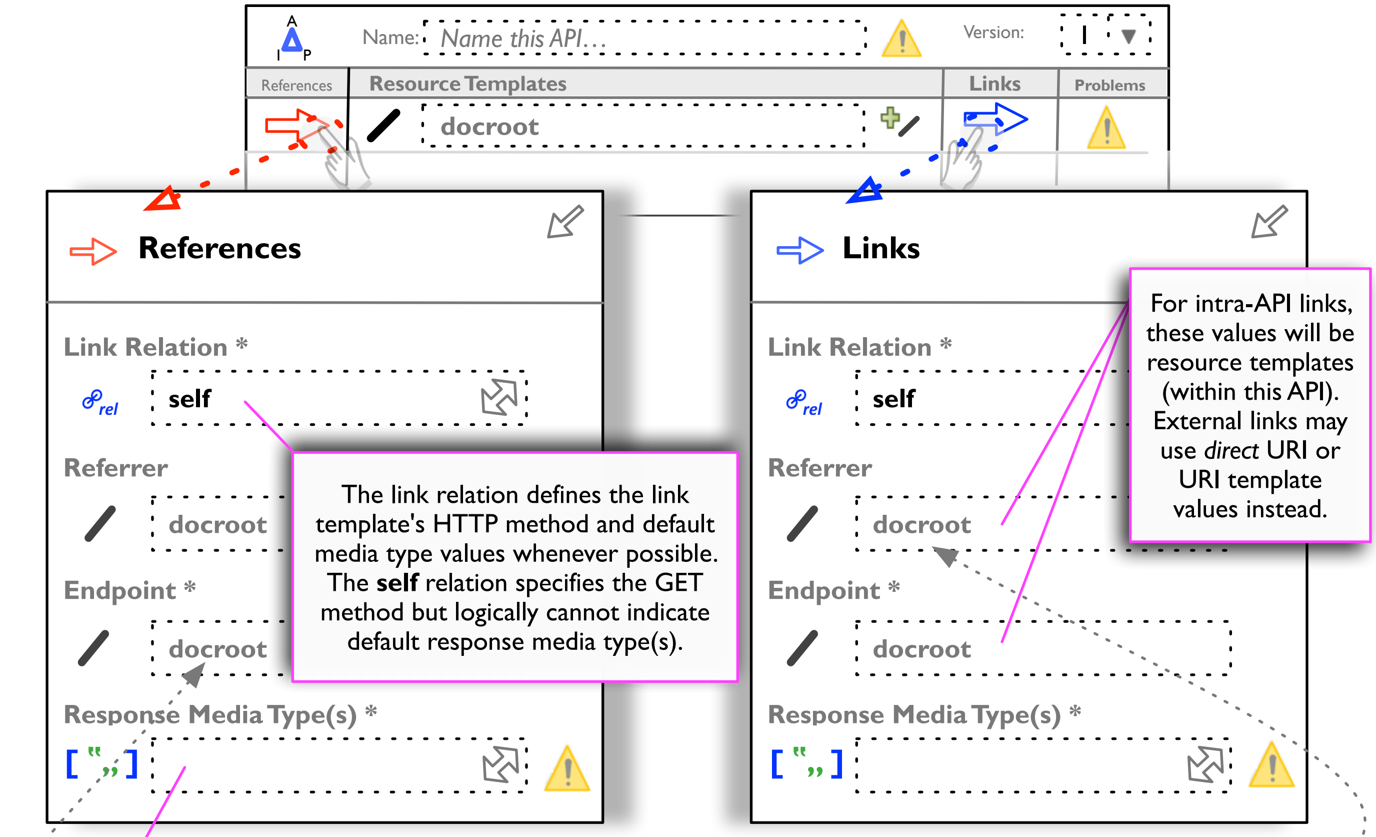
How? (continued)

6 *Representing the Docroot*

The **docroot** has a problem.



It's default (tool-generated) *self* **Link Template,** which uses HTTP GET, does not specify any response representation media types. There is *nothing* to GET from the docroot. Here is how we can change that:



The link relation defines the link template's HTTP method and default media type values whenever possible. The **self** relation specifies the GET method but logically cannot indicate default response media type(s).

For intra-API links, these values will be resource templates (within this API). External links may use *direct* URI or URI template values instead.

*The two ends of a single **Link Template**. Note that the **self** link relation is intended to loop back, meaning referrer = endpoint, but other link relations may point elsewhere via URI, URI Template, or Resource Template (within the same or different API).*

The response media type(s) value is a comma-separated list of media type values. Any valid HTTP media type (e.g. *application/json* or *image/jpeg*) may be included in this list, but, for serialized data model representation, ***application/wrml*** is preferred for its ability to specify both **Format** and **Schema**.

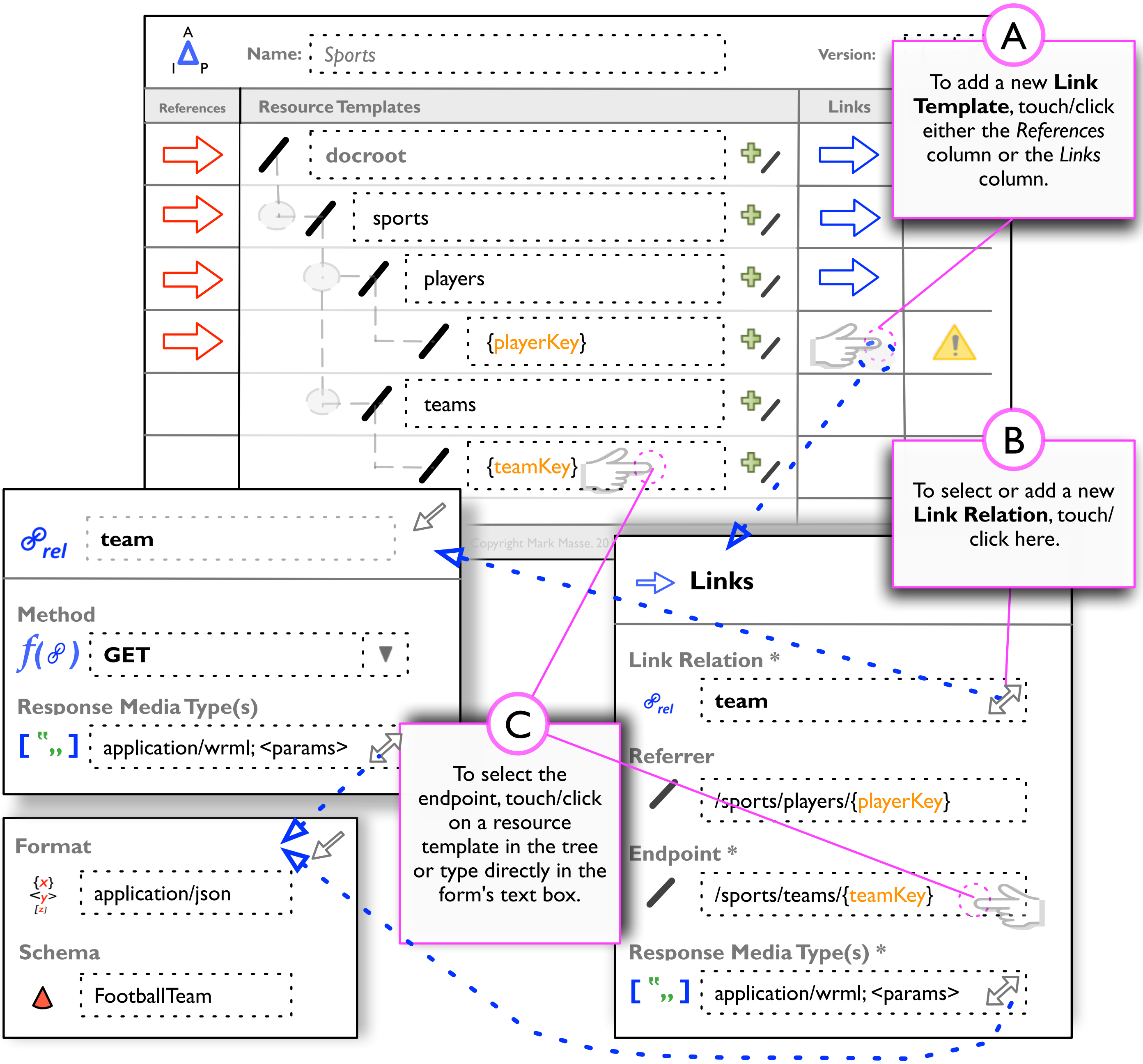
REST API Design with WRML

How? (continued)

7 *Linking*

To link all FootballPlayer models originating from .../{playerKey} to their respective teams, we need a *team* **Link Template** from the .../{playerKey} resource template to the .../{teamKey} resource template. Then, at runtime, an individual FootballPlayer model's teamKey field value completes its *team* link.

This approach ensures that there is no need to store/encode/decode an API's URIs within clients or back-end services. WRML's runtime uses API metadata to translate URIs to application domain-defined keys and *vice versa*.



Name: Sports

Version:

docroot

sports

players

{playerKey}

teams

{teamKey}

rel team

Method

GET

Response Media Type(s)

application/wrml; <params>

Format

application/json

Schema

FootballTeam

Links

Link Relation *

team

Referrer

/sports/players/{playerKey}

Endpoint *

/sports/teams/{teamKey}

Response Media Type(s) *

application/wrml; <params>

To add a new **Link Template**, touch/click either the *References* column or the *Links* column.

To select or add a new **Link Relation**, touch/click here.

To select the endpoint, touch/click on a resource template in the tree or type directly in the form's text box.

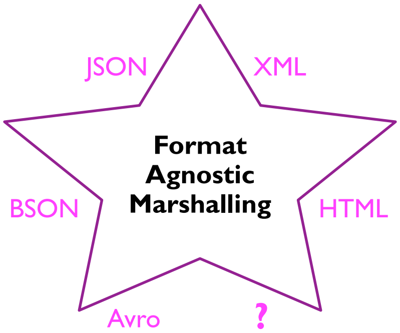
Schema Design with WRML

Introduction

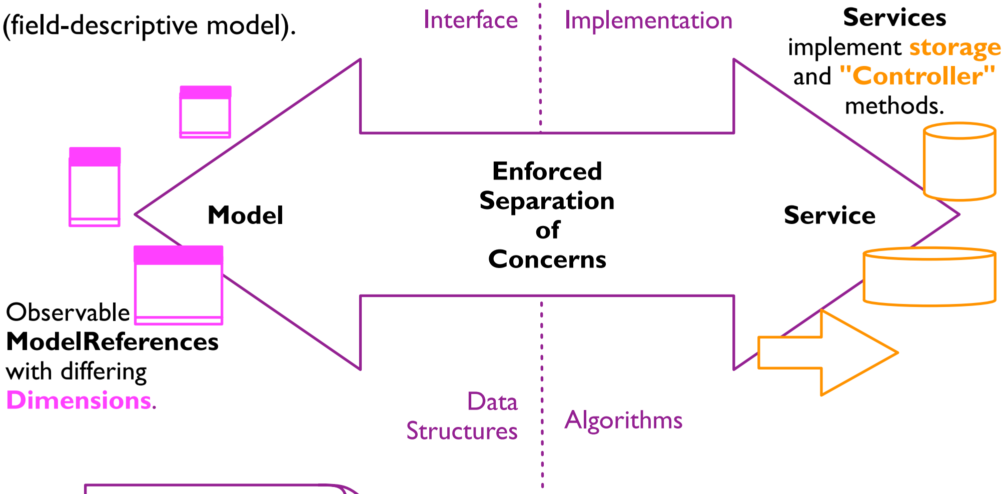
WRML represents resource state using schemas so that REST APIs (and their clients) can easily identify, implement, and document an application's data model "types". The illustration below outlines some of the advantages to using the WRML approach to ***data modeling***.

A WRML **Schema**:

- Exists within a namespace (e.g. org/wrml/model).
- *May* identify any number of "base" schemas.
- **Must** embed at least one **Field** (field-descriptive model).



Format Agnostic Marshalling



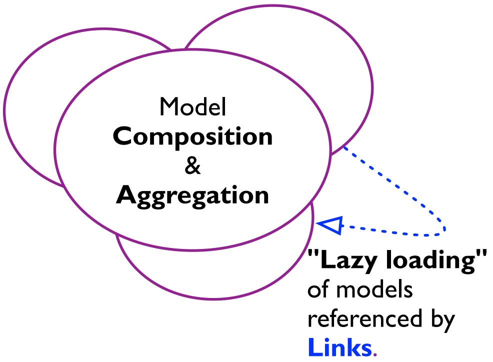
Services implement **storage** and **"Controller"** methods.

Model

Enforced Separation of Concerns

Service

Observable **ModelReferences** with differing **Dimensions**.



Model **Composition & Aggregation**

"Lazy loading" of models referenced by **Links**.

Java Class File

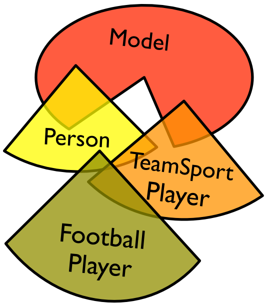
Automatic generation of Java interface (bytecode) representations of WRML schemas provides Java service developers a natural (static) POJO interface for each schema.

Web-based Modeling Tools

Automatic viewing and editing of models and their schemas via WRML-based tools.

"Editable Javadoc"

WRML's **Model Heap** creates and caches model instances "sub-atomically" by ***sharding*** their field value storage by schema id.



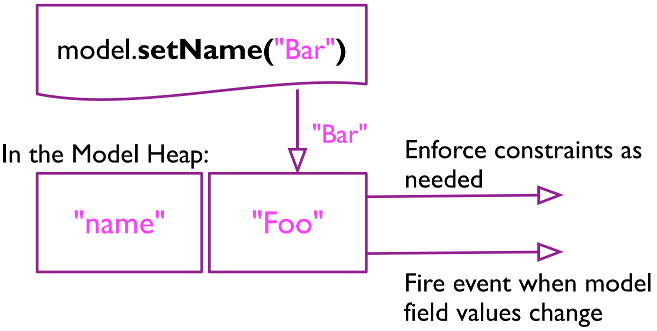
Built-in model "look-ups" using Schema-defined **key** field values.

{playerKey}

Model singularity (instance folding) with field-level data storage shared between references to the same modeled data.

WRML provides built-in "**MVC models**" with constrainable and observable fields:

In app code:



```
model.setName("Bar")
```

In the Model Heap:

Enforce constraints as needed

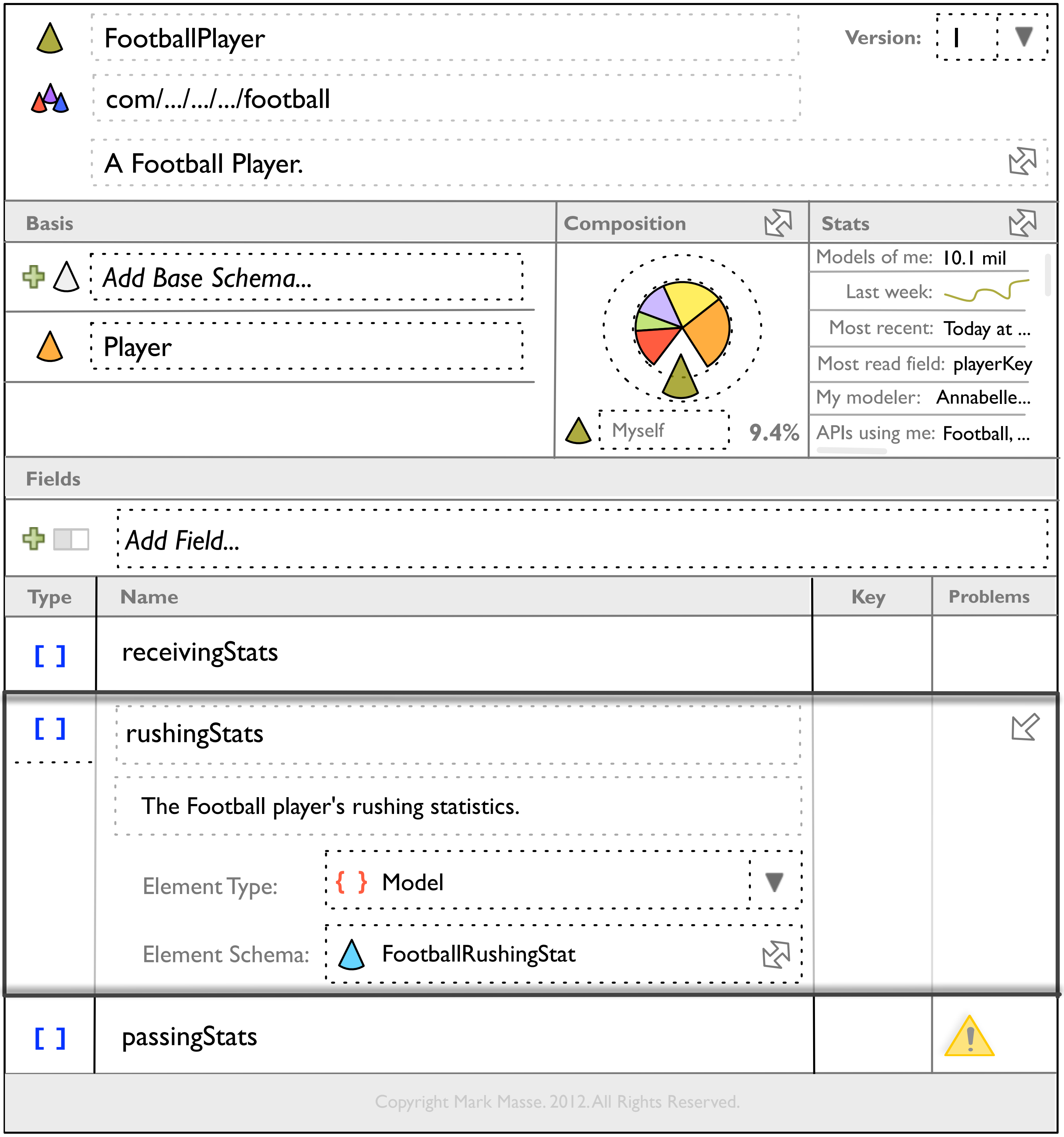
Fire event when model field values change

Schema Design Tool

Schema Design with WRML

Schemaboard

"Schemaboard" is the codename of the WRML tool/screen that provides the schema design screen concept illustrated below.



FootballPlayer

Version: 1 ▼

com/.../.../.../football

A Football Player.

Basis

✚ *Add Base Schema...*

Player

Composition

Myself 9.4%

Stats

Models of me: 10.1 mil

Last week:

Most recent: Today at ...

Most read field: playerKey

My modeler: Annabelle...

APIs using me: Football, ...

Fields

✚ *Add Field...*

Type	Name	Key	Problems
[ ]	receivingStats		
[ ]	rushingStats The Football player's rushing statistics. Element Type: { } Model ▼ Element Schema: FootballRushingStat		
[ ]	passingStats		⚠

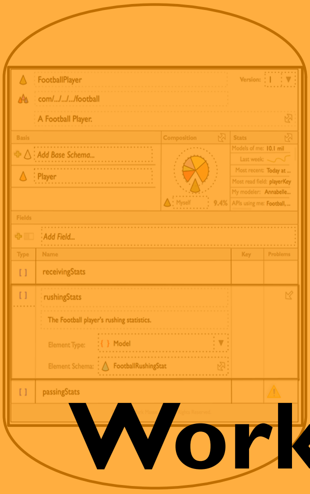
Schema Design Tool

Schema Design with WRML

How?

1 Search by tags (e.g. "Football Player") for an existing schema that may meet your data modeling needs. The goal here is to facilitate the development of reusable algorithms that know how to manipulate models conforming to a given schema (e.g. **Services**).

2 Create a new schema with a name of your choosing.



FootballPlayer Version: 1 ▼
com/.../.../.../football
A Football Player.

Basis: Add Base Schema... | Player

Composition: Myself 9.4%

Stats:
Models of me: 10.1 mil
Last week:
Most recent: Today at ...
Most read field: playerKey
My modeler: Annabelle...
APIs using me: Football, ...

Fields: Add Field...

Type	Name	Key	Problems
[ ]	receivingStats		
[ ]	rushingStats The Football player's rushing statistics. Element Type: { } Model ▼ Element Schema: FootballRushingStat		
[ ]	passingStats		⚠

Work - in - Progress

LinkRelation
A unique, perhaps reusable link specification; like a sharable "function signature".

Format
A serialization (aka "wire") format for WRML data.

Schema
Describes the structure of a certain category of model.

Collection (of Schema)
Contains a multitude of **Schema**.

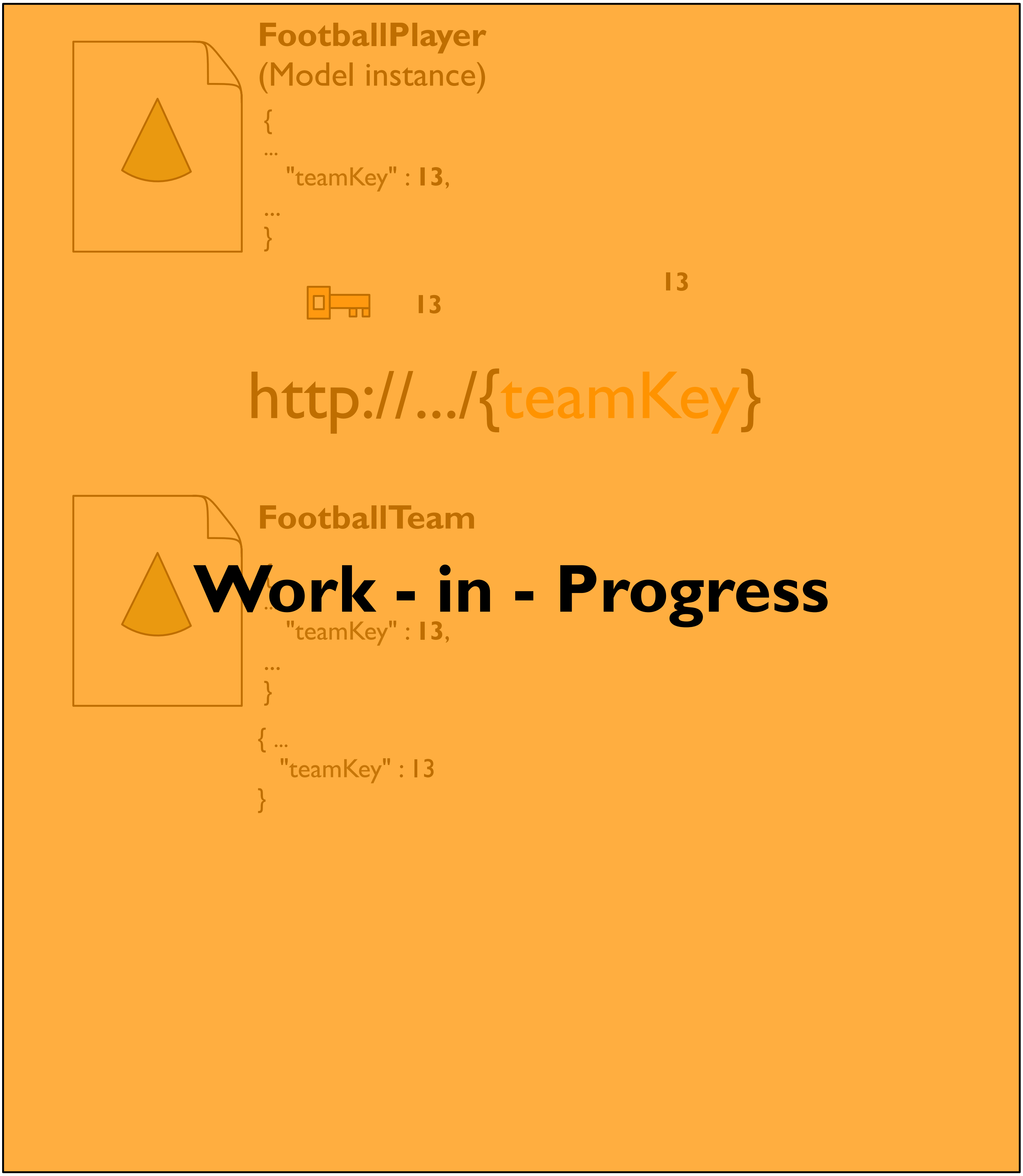
Constraint
Applies some limitation to the **Field** or **Schema** that it is associated with.

Icon	Name
[ ]	
{ }	
🔑	

Link
A hypermedia reference (with an identified **LinkRelation**) embedded within a **Document** (in a field value).

Field
Embedded within a **Schema** to describe a field that may be found in any conforming model.

Keys



FootballPlayer

(Model instance)

```
{
...
   "teamKey" : 13,
...
}
```

13

13

http://.../{teamKey}

FootballTeam

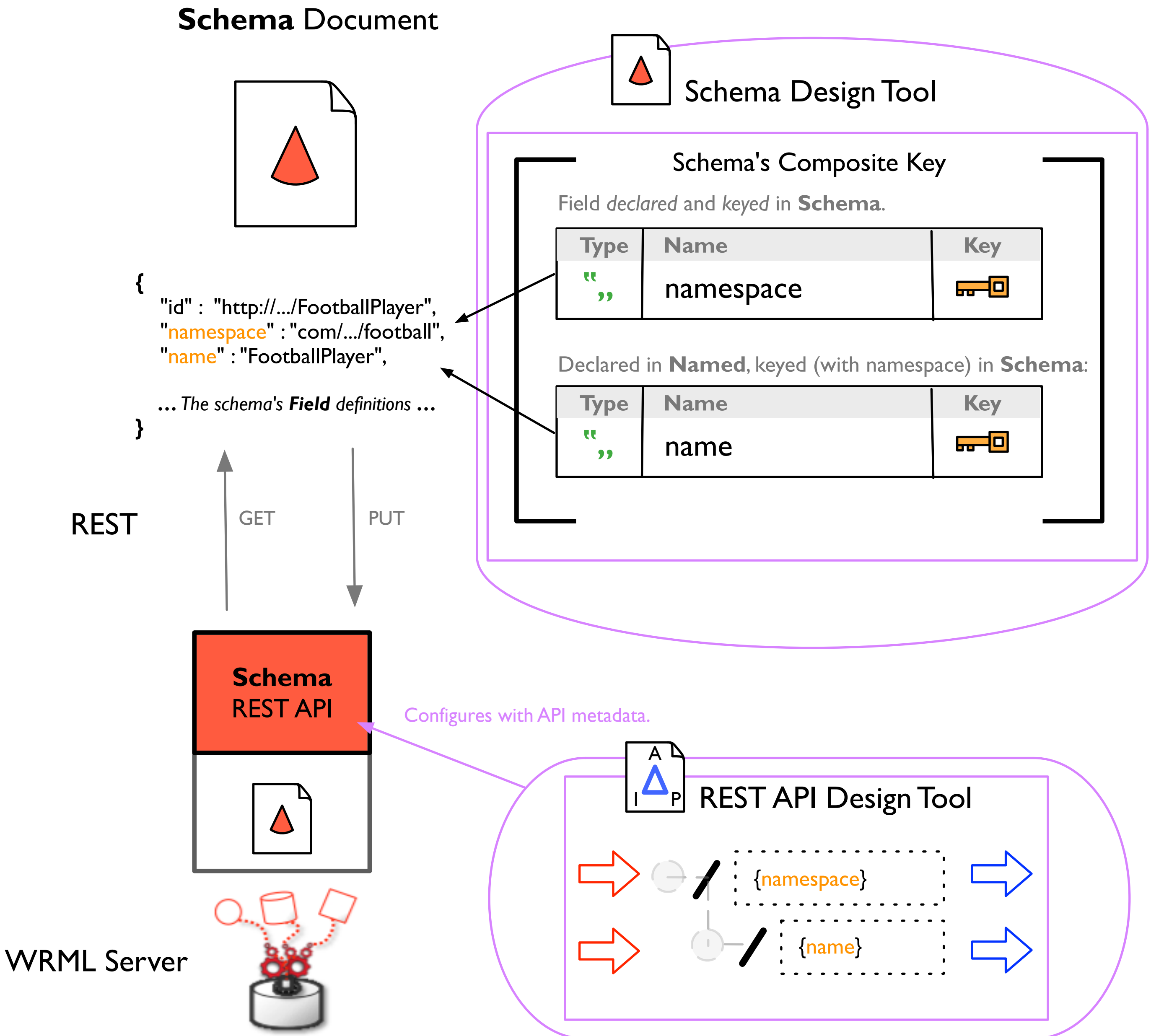
Work - in - Progress

```
{
...
   "teamKey" : 13,
...
}
```

```
{ ...
   "teamKey" : 13
}
```

Schema REST API Design

Like other languages with schema-based type systems, each WRML Schema has a **name** that is unique within a **namespace**. Schema name's are mixed uppercase. Schema namespaces are all lowercase with forward slashes ('/') used to separate their hierarchical components/segments.



Schema Document

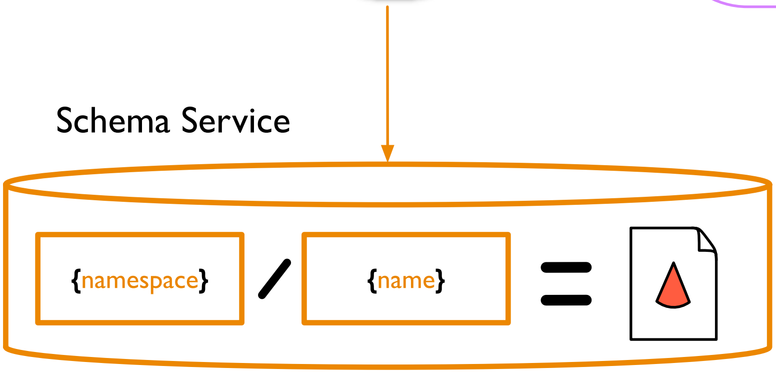
```
{
   "id" :  "http://.../FootballPlayer",
   "namespace" : "com/.../football",
   "name" : "FootballPlayer",

   ... The schema's Field definitions ...
}
```

Field *declared* and *keyed* in **Schema**.

Type	Name	Key
" "	namespace	🔑

Declared in **Named**, keyed (with namespace) in **Schema**:

Type	Name	Key
" "	name	🔑



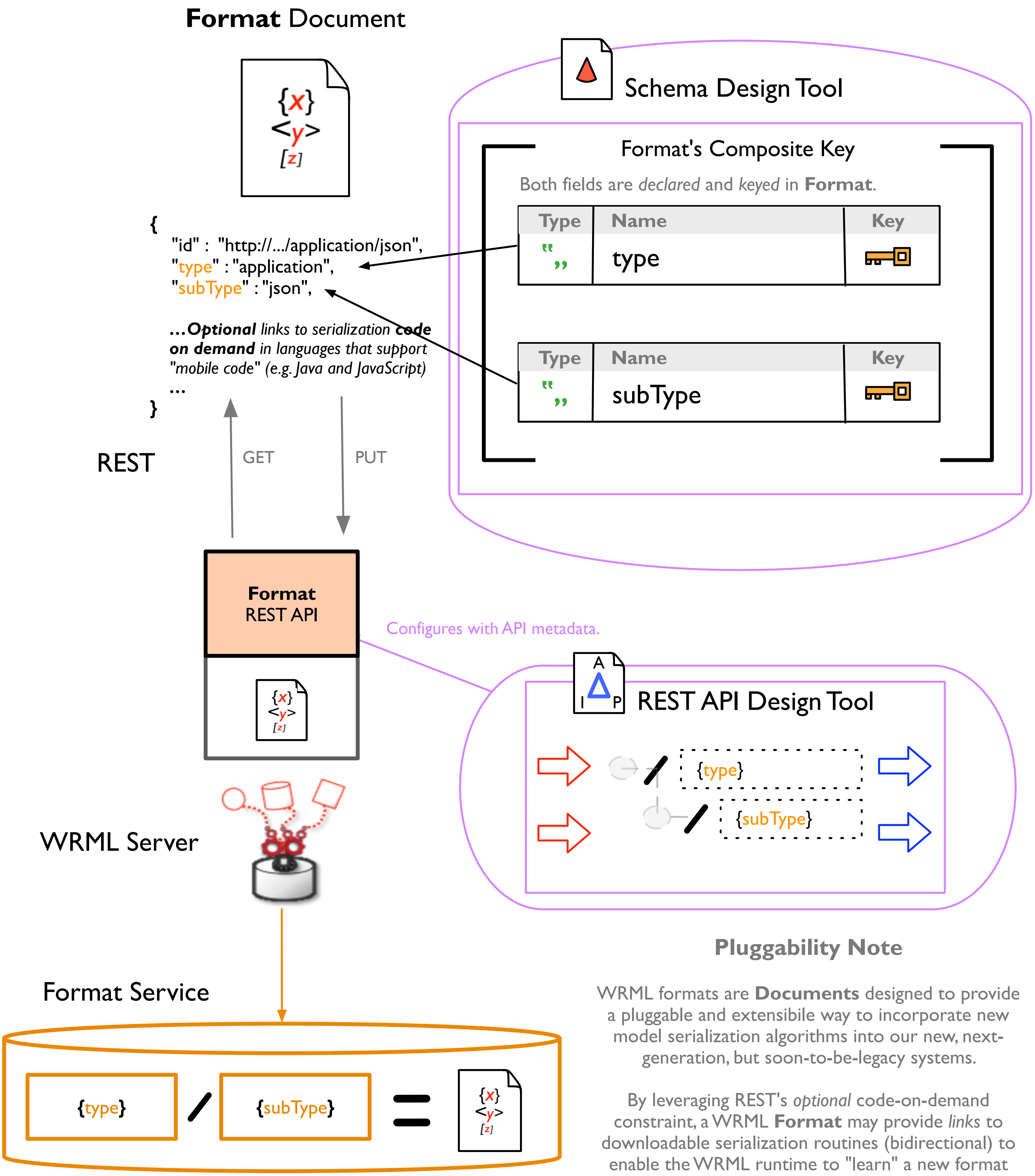
Bootstrapping Note

WRML schemas are **Documents** with embedded **Field** models that describe the runtime and serialization characteristics of a WRML model's fields.

Defining WRML's **Schema** schema (metaschema) as an extension of the **Document** schema enables WRML to manage Schema's using a Schema Design Tool and Schema REST API, both of which are built using WRML.

Format REST API Design

Following the convention established by HTTP's media types, a **Format** may be uniquely identified by its **type/subtype** hierarchy pair.



Format Document

```
{
   "id" :  "http://.../application/json",
   "type" : "application",
   "subType" : "json",

   ...Optional links to serialization code
   on demand in languages that support
   "mobile code" (e.g. Java and JavaScript)
   ...
}
```

REST

Format
REST API

WRML Server

Format Service

Schema Design Tool

Format's Composite Key

Both fields are *declared* and *keyed* in **Format**.

Type	Name	Key
“ ”	type	🔑

Type	Name	Key
“ ”	subType	🔑

Configures with API metadata.

REST API Design Tool

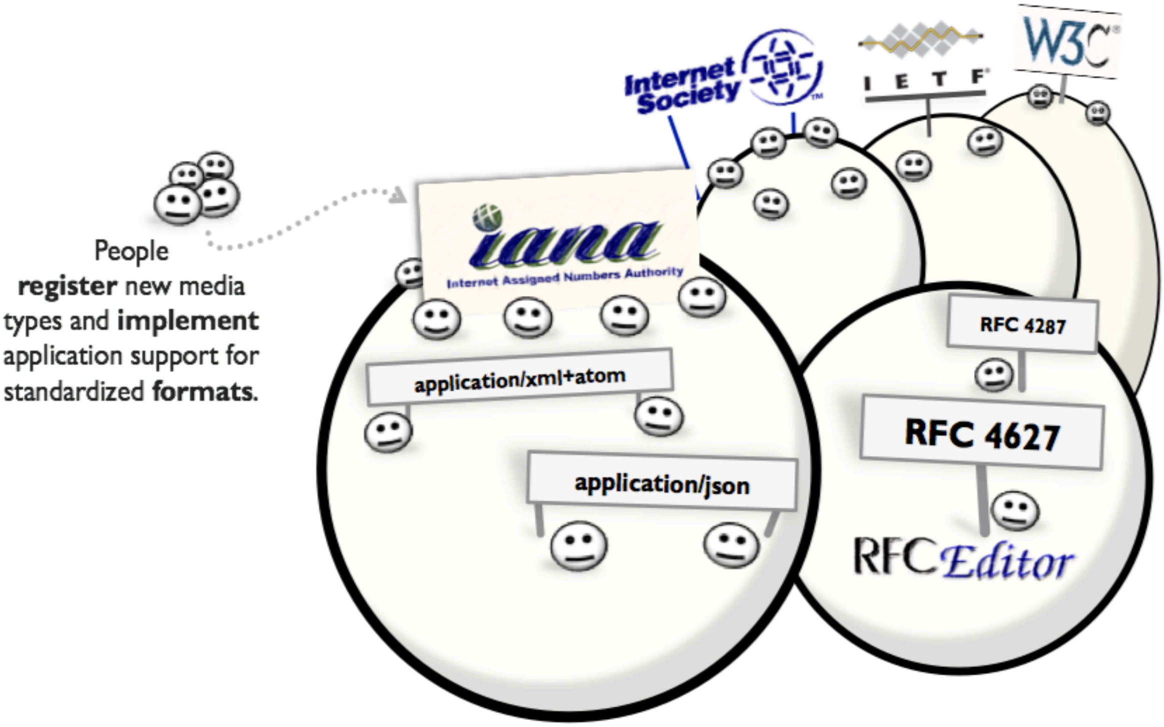
Pluggability Note

WRML formats are **Documents** designed to provide a pluggable and extensibile way to incorporate new model serialization algorithms into our new, next-generation, but soon-to-be-legacy systems.

By leveraging REST's *optional* code-on-demand constraint, a WRML **Format** may provide *links* to downloadable serialization routines (bidirectional) to enable the WRML runtime to "learn" a new format on-the-fly.

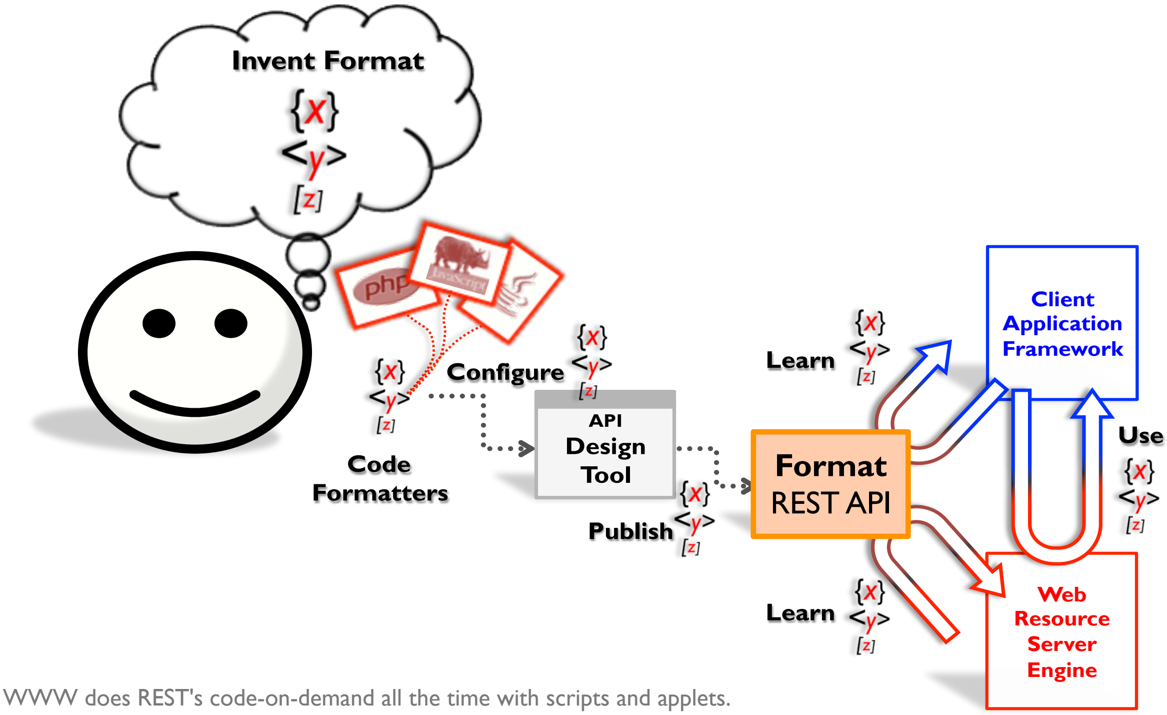
WRML Media Type - Pluggable Formats

Model serialization without **application/wrml** ...



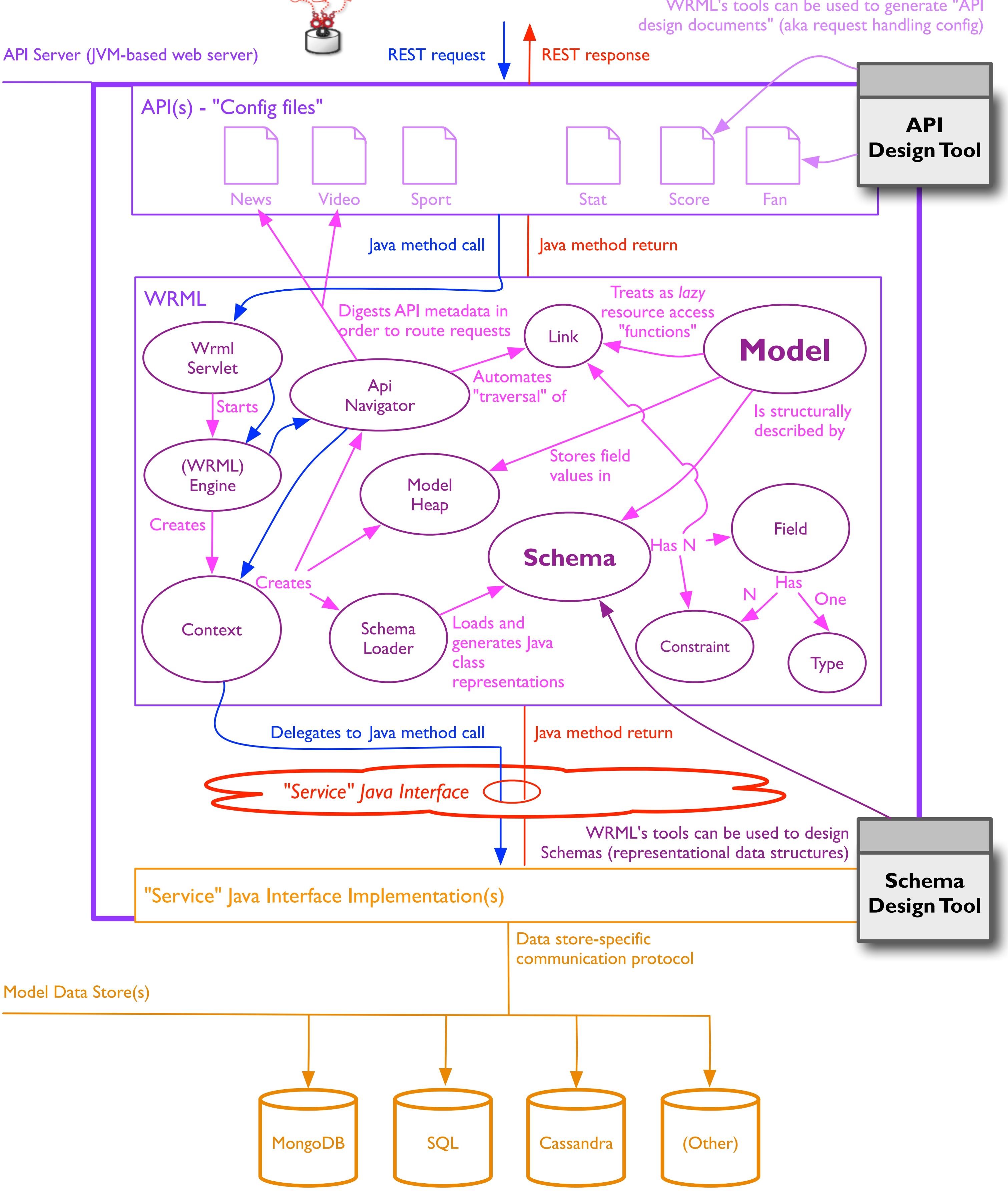
People **register** new media types and **implement** application support for standardized **formats**.

Model serialization with **application/wrml** ...



Note that the WWW does REST's code-on-demand all the time with scripts and applets.

WRML Server Internals



WRML's tools can be used to generate "API design documents" (aka request handling config)

API Server (JVM-based web server)

REST request

REST response

API(s) - "Config files"

News

Video

Sport

Stat

Score

Fan

API Design Tool

Java method call

Java method return

WRML

Wrml Servlet

Starts

(WRML) Engine

Creates

Context

Api Navigator

Digests API metadata in order to route requests

Automates "traversal" of

Link

Treats as *lazy* resource access "functions"

Model

Is structurally described by

Stores field values in

Model Heap

Creates

Schema Loader

Loads and generates Java class representations

Schema

Has N

Field

Has

N

One

Constraint

Type

Delegates to Java method call

Java method return

"Service" Java Interface

WRML's tools can be used to design Schemas (representational data structures)

Schema Design Tool

"Service" Java Interface Implementation(s)

Data store-specific communication protocol

Model Data Store(s)

MongoDB

SQL

Cassandra

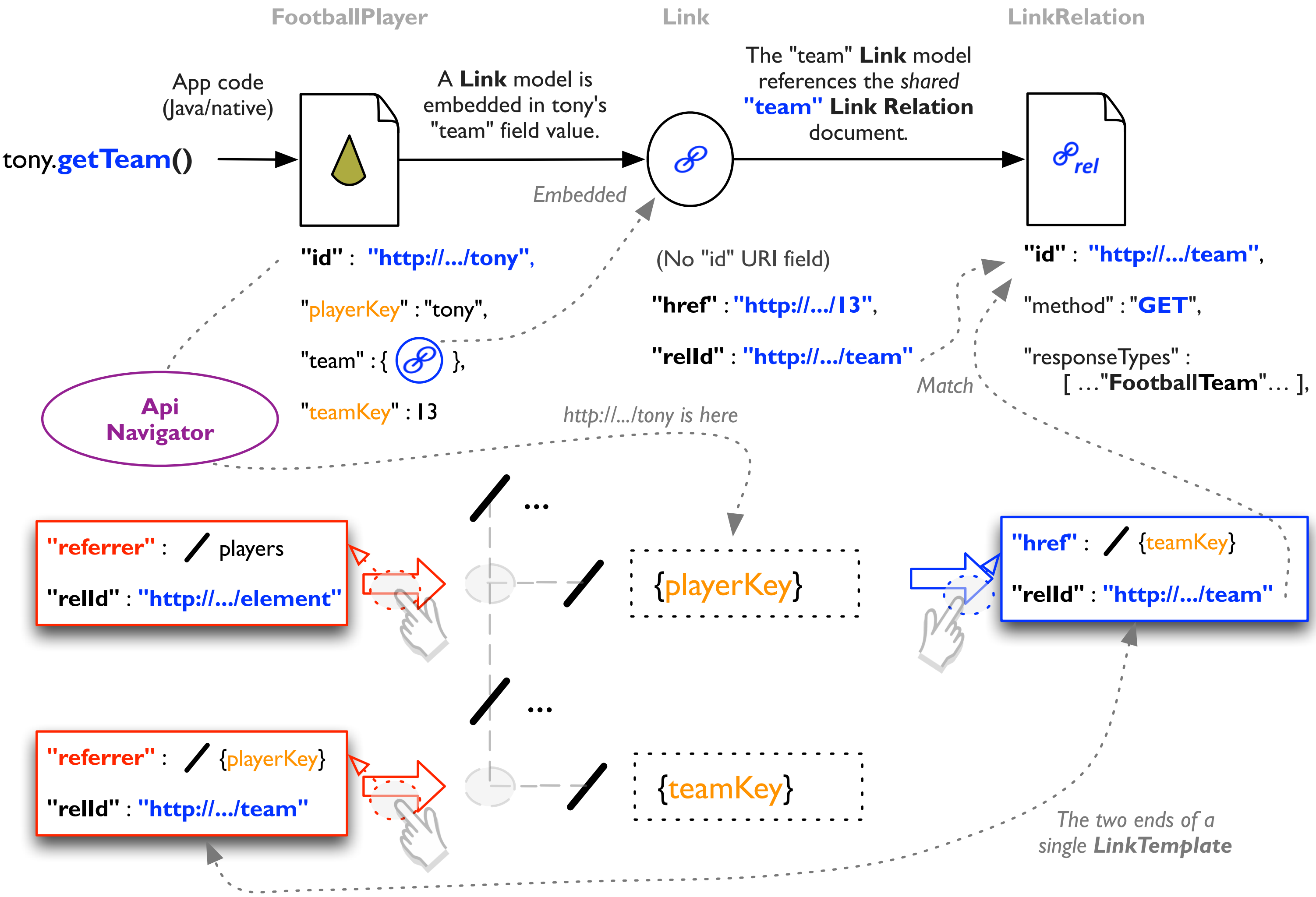
(Other)

Api Navigation

The WRML runtime guarantees that every saved **Document** model will have a non-null URI **"id"**.

The WRML runtime ensures that every "saved" Document (a model representing a REST API-managed resource) has a non-null **"id"** field value which uniquely identifies it with a URI-based *key*.

The example below follows the **"team" Link** from a **FootballPlayer** identified as **"http://<Absolute URI>/tony"**.



FootballPlayer

Link

LinkRelation

App code (Java/native)

tony.**getTeam()**

A **Link** model is embedded in tony's "team" field value.

Embedded

The "team" **Link** model references the *shared* **"team" Link Relation** document.

"id" : **"http://.../tony",**

"playerKey" : "tony",

"team" : { },

"teamKey" : 13

(No "id" URI field)

"href" : **"http://.../13",**

"relId" : **"http://.../team"**

Match

"id" : **"http://.../team",**

"method" : **"GET",**

"responseTypes" :
[..."**FootballTeam**"...],

Api Navigator

http://.../tony is here

"referrer" : / players

"relId" : **"http://.../element"**

/ ...

{playerKey}

"href" : / {teamKey}

"relId" : **"http://.../team"**

/ ...

"referrer" : / {playerKey}

"relId" : **"http://.../team"**

{teamKey}

*The two ends of a single **LinkTemplate***

The WRML runtime *maps* each saved Document model to its origin **Resource.**

The WRML runtime ensures that every "saved" Document may be associated with a single, configured **Api's ResourceTemplate**.

WRML client's are naturally configured to know about the Apis that they connect with for data and controls. On the client-side, WRML's internal **ApiNavigator** digests the Api-modeled metadata to *implement* the Links between Documents. From a Document model instance's perspective, the Api's metadata is like a shopping mall's store directory with a "*you are here*" sign to indicate which ResourceTemplate (URI path) is its point of origin (*matching* its **"id"**).

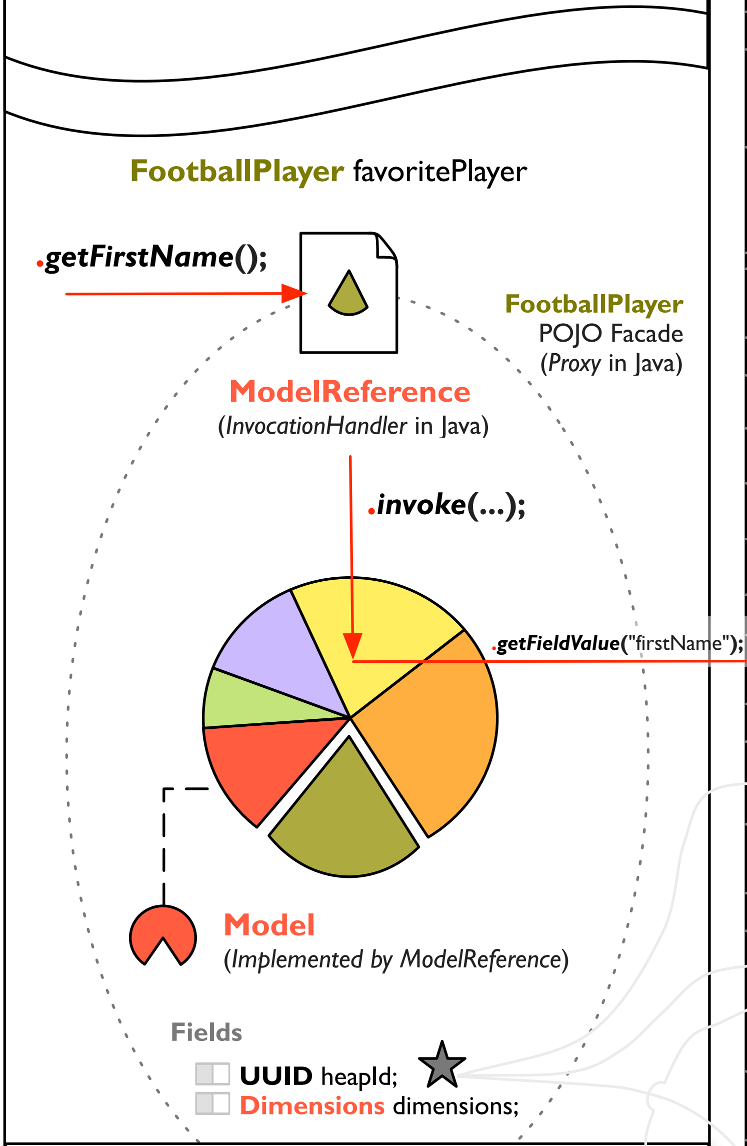
WRML servers are configured to know all of the Apis that they are implementing/hosting. On the server-side, WRML's ApiNavigator component digests the Api-modeled metadata to *route REST API requests* for Documents (and implement the HTTP OPTIONS method's response). A WRML server is responsible for ensuring that all Document **"id"** field values and Link **"href"** field values are written out in responses with absolute URI values.

Additionally, the ApiNavigator ensures that the **"href"** values are updated to reflect changes to URI template parameterized field values (HATEOAS on both client and server-side). So on both sides of a client-server app's communication, there are good reasons for the WRML runtime to utilize the Api configuration metadata that is generated from a tool like **API Binder**.

WRML's Schema-Sharded Model Heap

App Code

```
FootballPlayer favoritePlayer;

favoritePlayer = context.get("tony", dimensions);

String firstName = favoritePlayer.getFirstName();
```



Note:

Other **ModelReference** instances, with either the same or different **Dimensions**, may share the *same* heapId value to enable direct field value sharing between *alternate* models of the "same data".

ModelHeap

Behind the scenes, WRML's in-memory **ModelHeap** divides a model's field values (representational state) into table-like **Shards**; one for each schema. A **ModelReference's heapId** connects it to its field values stored within the sharded heap.

The heap uses WRML's schematic keys to ensure model uniqueness (singularity), and enable heap-based model look-ups (cache get) based on schema-designed key field values.

Cacheable

Heap ID	Name	Value	Key
★	cacheTag	"XYZ-123"	
	secondsToLive	43200	
◆	cacheTag	"SMS-21"	

Document

Heap ID	Name	Value	Key
★	id	"http://.../tony"	🔑
⬟	id	"http://.../13"	🔑
◆	id	"http://.../4"	🔑
⯃	id	"http://.../21"	🔑

Person

Heap ID	Name	Value	Key
★	firstName	"Tony"	

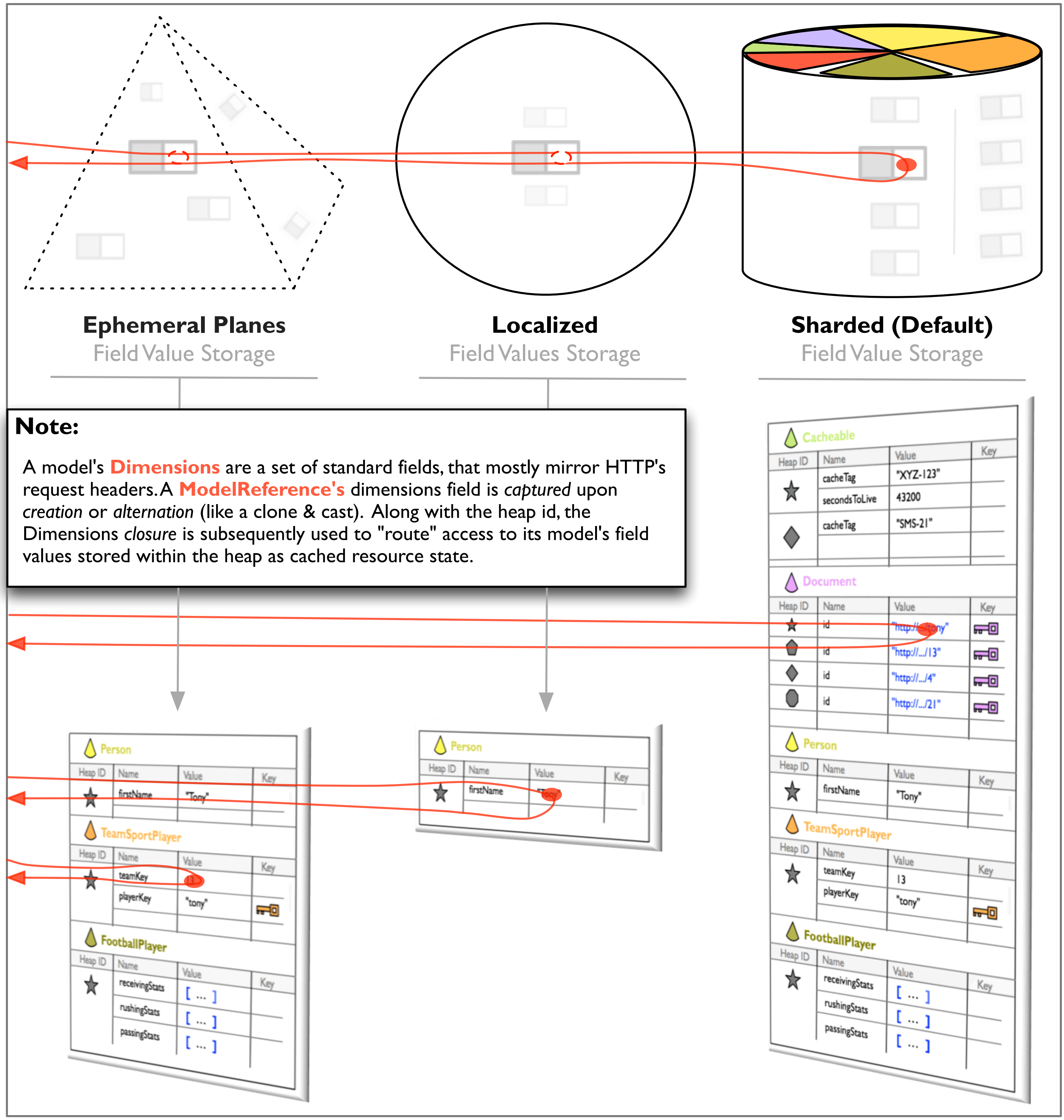
TeamSportPlayer

Heap ID	Name	Value	Key
★	teamKey	13	
	playerKey	"tony"	🔑

FootballPlayer

Heap ID	Name	Value	Key
★	receivingStats	[…]	
	rushingStats	[…]	
	passingStats	[…]	

ModelHeap



Ephemeral Planes
Field Value Storage

Localized
Field Values Storage

Sharded (Default)
Field Value Storage

Note:

A model's **Dimensions** are a set of standard fields, that mostly mirror HTTP's request headers. A **ModelReference's** dimensions field is *captured* upon *creation* or *alternation* (like a clone & cast). Along with the heap id, the Dimensions *closure* is subsequently used to "route" access to its model's field values stored within the heap as cached resource state.

Person

Heap ID	Name	Value	Key
★	firstName	"Tony"	

TeamSportPlayer

Heap ID	Name	Value	Key
★	teamKey	13	
	playerKey	"tony"	🔑

FootballPlayer

Heap ID	Name	Value	Key
★	receivingStats	[...]	
	rushingStats	[...]	
	passingStats	[...]	

Person

Heap ID	Name	Value	Key
★	firstName	"tony"	

Cacheable

Heap ID	Name	Value	Key
★	cacheTag	"XYZ-123"	
	secondsToLive	43200	
◆	cacheTag	"SMS-21"	

Document

Heap ID	Name	Value	Key
★	id	"http://.../tony"	🔑
⬟	id	"http://.../13"	🔑
◆	id	"http://.../4"	🔑
⯃	id	"http://.../21"	🔑

Person

Heap ID	Name	Value	Key
★	firstName	"Tony"	

TeamSportPlayer

Heap ID	Name	Value	Key
★	teamKey	13	
	playerKey	"tony"	🔑

FootballPlayer

Heap ID	Name	Value	Key
★	receivingStats	[...]	
	rushingStats	[...]	
	passingStats	[...]	

Internally, the **ModelHeap** separates a model's field values (representational state) based upon the its **Dimensions**. The heap uses a model's Dimensions to guide field value accessors to the appropriate (internal memory) storage location.

The **Ephemeral Plane's** storage location may be used to hold temporary edits to a model's field values (e.g. prior to an update "commit"). When accessed from a model that is dimensioned to be ephemeral, any writable field values written here will "override" the *real* values stored in either the localized or standard (default locale) sharded storage location. A model dimensioned with a (non-default) **Locale** value, will store its localizable field values in the corresponding locale-specific storage location within the heap.

Service Wiring

Services register for request delegation by expressing the model Dimensions (e.g. media type and locale) and key variable values that describe/encompass the models that they own.

Collection state fulfillment and searching

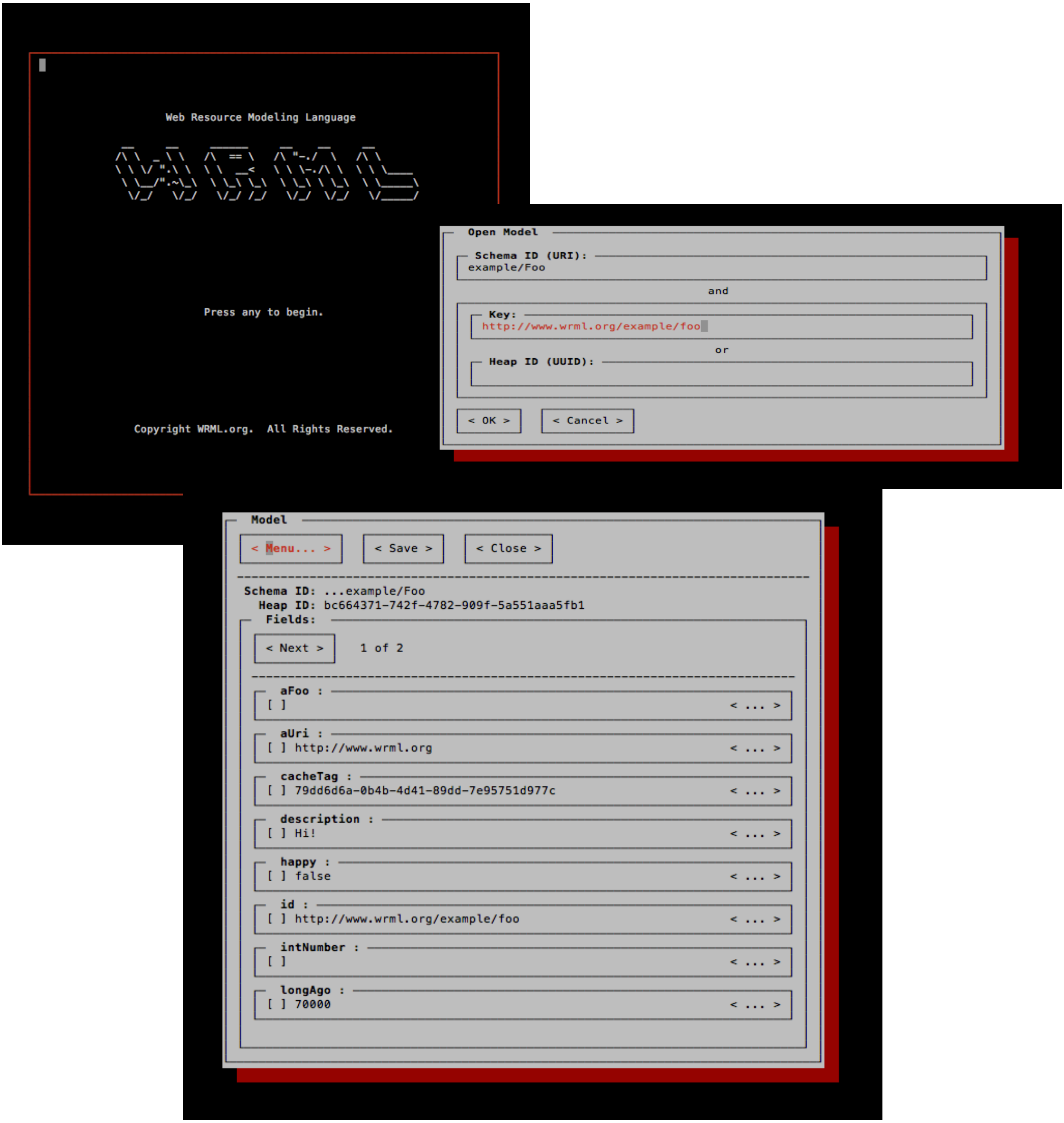
Work - in - Progress

Delegated invocation of "Controller" methods.

The built-in "generic" model viewer demonstration

WRML ships with a "Terminal" or "command line" graphical user interface (CLGUI) that supports some basic modeling operations. The screen shots below demonstrate the **WRML CLI's GUI**.

```
> java org.wrml.cli.Wrml wrml.json
```



```
Web Resource Modeling Language

Press any to begin.

Copyright WRML.org.  All Rights Reserved.
```

```
Open Model
  Schema ID (URI):
  example/Foo
                                   and
    Key:
    http://www.wrml.org/example/foo
                                   or
    Heap ID (UUID):

  < OK >   < Cancel >
```

```
Model
  < Menu... >   < Save >   < Close >
  ------------------------------------------------------------------
  Schema ID: ...example/Foo
    Heap ID: bc664371-742f-4782-909f-5a551aaa5fb1
     Fields:
    < Next >    1 of 2
    ----------------------------------------------------------------
    aFoo :
    [ ]                                                    < ... >
    aUri :
    [ ] http://www.wrml.org                                < ... >
    cacheTag :
    [ ] 79dd6d6a-0b4b-4d41-89dd-7e95751d977c               < ... >
    description :
    [ ] Hi!                                                < ... >
    happy :
    [ ] false                                              < ... >
    id :
    [ ] http://www.wrml.org/example/foo                    < ... >
    intNumber :
    [ ]                                                    < ... >
    longAgo :
    [ ] 70000                                              < ... >
```