SemantEco: A Modular Framework for Integration of Ecological, Environmental, and Earth Sciences Data

Evan W. Patton
Tetherless World Constellation
Rensselaer Polytechnic Institute
110 8th Street
Troy, NY 12180, USA
pattoe@rpi.edu

Patrice Seyed
University of New Mexico
1 University Boulevard
Albuquerque, NM 87131, USA
seyeda2@rpi.edu

Deborah L. McGuinness
Tetherless World Constellation
Rensselaer Polytechnic Institute
110 8th Street
Troy, NY 12180, USA
dlm@cs.rpi.edu

May 16, 2013
Abstract

In this paper we discuss the architecture of SemantEco, a modular framework for integrating data from ecological, environmental, and Earth sciences. The architecture of SemantEco encapsulates features as modules that can be combined to present data in different ways. We provide an in-depth discussion on module implementation as well as detail the existing modules available in SemantEco and how they interact with one another. The architecture is generalizable to other domains as well.
Acknowledgements

The Tetherless World Constellation is supported in part by Fujitsu, Lockheed Martin, LGS, Microsoft Research, Qualcomm, in addition to sponsored research from DARPA, IARPA, NASA, NSF, and USGS. Evan W. Patton is supported by an NSF Graduate Research Fellowship.
Licensing

SemantEco is made available under the Apache License v2.0. This documentation is made available under the terms of Creative Commons BY-SA 3.0 Unported License.
# Contents

| List of Figures | 3 |
| List of Tables  | 4 |
| 1 Introduction  | 5 |
| 1.1 Definitions | 6 |
| 1.2 Where To Begin | 6 |
| 2 Architecture Design | 7 |
| 2.1 The SemantEco Servlet | 7 |
| 2.1.1 Initialization of SemantEco on Deployment | 7 |
| 2.1.2 Special URL Patterns | 7 |
| 2.2 The Module Manager | 8 |
| 2.2.1 Installing Modules | 8 |
| 2.2.2 Uninstalling Modules | 8 |
| 2.2.3 Visiting Modules | 9 |
| 2.3 ModuleConfiguration | 9 |
| 2.4 The Query Executor | 10 |
| 2.5 Client-Server Communication | 11 |
| 2.5.1 Query Methods | 11 |
| 2.6 Debugging Techniques | 12 |
| 2.6.1 Web Sockets | 12 |
| 2.6.2 Remote Java Debugging | 12 |
| 3 Semantic Model | 13 |
| 3.1 Sites, Measurements, and Pollution | 13 |
| 3.2 Modeling Regulation Violations | 13 |
| 3.3 Querying the Model | 13 |
| 4 User Interface | 14 |
| 4.1 The Semantic Entry Point: index.jsp | 14 |
| 4.1.1 The SemantEco Tag Library | 14 |
| 4.2 Map Interface | 15 |
| 4.3 Module Facets | 15 |
| 4.4 Hierarchical Search with JSTree | 15 |
| 4.5 Charting with jqPlot | 15 |
List of Figures
Chapter 1

Introduction

In many places around the world, wildlife and the habitats on which they depend are deteriorating. For instance, almost 40 percent of the United States’ freshwater fish species are considered at risk or vulnerable to extinction according to [1]. Aiming at preserving the environment and wildlife, scientists and resource managers have initiated various efforts to monitor ecological and environmental trends, investigate causes and possible effects of pollution, and identify threats to wildlife and their habitats. To be effective, these efforts must be undertaken with a large scale and multidisciplinary perspective. However, by nature this approach requires access and use of disparate data sources and systems. For example, the USGS provides integrated science and technology to support resource managers in the U.S. Department of the Interior (DOI) through initiatives such as the Wyoming Landscape Conservation Initiative (WLCI): an effort to assess and enhance aquatic and terrestrial habitats at a landscape scale in southwest Wyoming. Decision support systems are one end result of scientific research that facilitates examination of the many tradeoffs and conflicting drivers that resource managers often wade through in their work, from energy and agricultural development to fish and wildlife conservation to recreational uses of public lands.

Meanwhile, semantic science and technologies have been used in environmental monitoring to facilitate knowledge encoding, data integration and collaborative scientific workflows [2]. Previously we proposed the Tetherless World Constellation Semantic Ecology and Environment Portal (SemantEco) [3,4] as both an environmental portal application and as an example of a semantic infrastructure for environmental informatics applications. The portal captures the semantics of domain knowledge using a family of ontologies, integrates environmental data (e.g. water quality measurement and regulatory data), infers pollution events using OWL2 inference, and leverages provenance. In this paper, we extend the focus of SemantEco beyond water quality and related human health effects to a more comprehensive effort that includes fish and wildlife species and their related health effects. As such, this extension is in concert with the One Health concept, defined as “the collaborative effort of multiple disciplines working locally, nationally, and globally to attain optimal health for people, animals and the environment” [5]. Consider a data manager interested in identifying correlations between declines of a bird population observed in a river due to chemicals released by nearby factories. There may be direct causal links (i.e. pollutant X in a diet kills Y) or indirect links (e.g. pollutant X causes reproductive system changes leading to lowered breeding success and resultant population declines). To identify these scenarios, resource managers need access to a number and variety of information and data sources including databases containing species observation counts, literature on the environmental effects of pollutants, how species may migrate and interact with one another, and so forth. Semantic technologies will facilitate access to multidisciplinary information that will aid resource managers in making decisions about complex ecosystems. These technologies also enhance reusability and address extensibility issues targeting challenges in the areas of data integration, provenance, and scalability.

We use semantic technologies to provide responses to these challenges. This work presents a new architecture that evolves an existing water quality portal into a general model for performing ecological and environmental data integration across many different data sources and domains. Additionally, this new architecture is designed to be horizontally scalable through the use of representational state transfer [6].
We also demonstrate how semantic technologies can be used to encourage interoperability by connecting one existing well-used observation schema - the Extensible Observation Ontology (OBOE) \[7\] to support interoperable observation data. We investigated the reasoning implications of adopting OBOE for interpreting data and found out that OBOE brings greater interoperability at the cost of longer reasoning time. Additionally, we now integrate various ecological and environmental data: wildlife observational data from the Avian Knowledge Network (AKN)\[1\] and U.S. Geological Survey (USGS)\[2\]; environmental criteria for wildlife from the Environmental Protection Agency (EPA)\[3\]; water body data from USGS; and health effects of contaminants on wildlife from Wildpro\[4\]. These are provided in addition to the existing water quality and regulatory data presented in earlier versions of this work. Our approach provides a formal encoding of the semantics of the data and provides services for automatic reasoning and visualizations. Furthermore, we compared the performance of a standard semantic web reasoner used in previous iterations with a new, customized rule based reasoner over our data. Lastly, we enhanced the provenance support of the existing portal by incorporating rationale as provenance. This extensibility enabled by semantic technologies allows the portal to consume data from many different domains and present it together for the purposes of informing environmental stakeholders.

1.1 Definitions

SemantEco is an amalgamation of many different ontologies and as such this technical report includes a number of detailed items. This section provides a number of definitions for common terms that readers will encounter throughout this document.

- **SemantEco Framework** - the conceptualization of SemantEco as a modular framework that combines different modules into a single application
- **SemantEco servlet** - a Java Servlet 3.0 Specification-compatible servlet that implements the SemantEco Framework
- **jQuery** - An extensible JavaScript library that provides powerful features for web application developers
- **AJAX** - Asynchronous Javascript and XML, a process by which web applications make calls to a server to obtain data while the user can continue to interact with the current web page
- **BBQ** - jQuery Back button and Query library; useful for providing navigation and state features in single-page web applications
- **RDF** - The Resource Description Framework defined by the World Wide Web Consortium
- **OWL** - The Web Ontology Language, in this work we make use of OWL 2 DL

1.2 Where To Begin

Readers interested in getting a general overview of how SemantEco is structured architectural should be able to read this document in order. Chapters are arranged such that having an understanding of earlier chapters provides a general background for future chapters. For readers interested in the application flow from a server perspective, Section \[2.1\] introduces the SemantEco servlet and how it is initialized server side. For readers interested in the client-side application flow, Section \[4.1\] introduces the SemantEco interface described in its index.jsp and walks through how it and the other user interface components operate.

\[1\]http://www.avianknowledge.net/
\[2\]http://www.usgs.gov/
\[3\]http://www.epa.gov/
\[4\]http://wildpro.twycrosszoo.org/
Chapter 2

Architecture Design

In this chapter we will discuss the architecture of the new SemantEco, from its adherence to the Java Servlet Specification so that it can be deployed in many different environments to its method of executing modules and queries to accomplish user tasks. We will also discuss how the RESTful client-server communication works to enable SemantEco both as a web application and as a web service.

2.1 The SemantEco Servlet

SemantEco is, at its core, a Java Servlet 3.0 servlet. As such, it can be deployed into a number of servlet containers, such as Tomcat, Jetty, and Glassfish. In this way, system administrators can easily deploy SemantEco alongside existing servlet-based services and reduce overall system maintenance. Previous versions of SemantEco used a built-in HTTP server and were thus less flexible for deployment as it would need to be proxied through a forward-facing web server such as Apache. While proxying is no longer necessary, it can still be beneficial for deployments where speed is important as a proxy can cache SemantEco’s responses.

The SemantEco servlet is, at its core, a ‘plug-and-play’ architecture. The servlet mostly provides glue code that is responsible for combining functionality of different modules. A module is a self-contained Java Archive that provides functionality to SemantEco via different methods. The specifics of how modules work will be discussed in Chapter 5.

2.1.1 Initialization of SemantEco on Deployment

When the servlet container starts or when a developer uses the maven tomcat7:deploy goal, the first action taken after the SemantEcoServlet is constructed is a call by the container to the init method per the Java Servlet 3.0 specification. During this initialization, SemantEco configures any missing directories, initializes its logging system (log4j), and sets up the module manager, which includes loading all prepackaged modules in the Web archive (war) file.

2.1.2 Special URL Patterns

The SemantEco responds to a number of URL patterns that require custom processing. These include:

- /js/config.js – Client configuration specific to the SemantEco deployment, including the base URL for the deployment and where to submit RESTful calls.

---

1 http://jcp.org/aboutJava/communityprocess/final/jsr315/index.html
2 http://tomcat.apache.org/
3 http://jetty.codehaus.org/jetty/
4 http://glassfish.java.net/
5 http://httpd.apache.org/
• /js/modules/* – Autogenerated JavaScript files to facilitate client-server communication (see Section 2.5).
• /log – WebSockets interface for sending debugging data to client.
• /rest/* – RESTful calls to SemantEco modules (see Section 2.5).

If a request is made to the servlet, these special URLs are handled in the servlet’s doGet method. Otherwise, the servlet container will attempt to retrieve the file named by the URL. If no such file exists than an HTTP 404 error will be generated.

2.2 The Module Manager

SemantEco modules are maintained by a class called the ModuleManager. The ModuleManager is responsible for loading modules from their JAR files and configuring them. If a module is removed from SemantEco, the ModuleManager will delete the module instance and unload the module files from the class path. The ModuleManager is also responsible for constructing the user interface and semantic models and modifying queries before they are executed. It accomplishes these tasks by consulting the available modules.

2.2.1 Installing Modules

ModuleManager uses the Apache Virtual File System to actively monitor the modules directory and respond to the addition or deletion of modules. During servlet initialization, the ModuleManager will actively scan the directory and fire addition events for each module available at that time to bootstrap the entire process. When a module is added the following steps occur:

1. The ModuleManager instantiates a ModuleClassLoader object responsible for reading in the JAR file and defining the classes in the JAR with the Java Virtual Machine. It also performs validation that at least one of the classes in the JAR represents a implementation of the Module interface.

2. The ModuleManager ‘explodes’ the JAR by extracting everything in the META-INF/res/ directory into the web application’s resources directory under a module-specific directory to prevent naming clashes between modules. This directory is named the same as the JAR without the ‘.jar’ extension.

3. The ModuleManager instantiates an instance of each Module implementation found in the jar.

4. The ModuleManager installs the newly created Module to its internal store.

5. The ModuleManager instantiates a ModuleConfiguration for each new module and loads the module.properties file into the configuration for the module if one exists. This configuration is then supplied to the module via its setModuleConfiguration method.

If during the initialization process an error occurs while processing the JAR’s classes the module will not be instantiated and the ModuleClassLoader will be destroyed. If the module throws an exception in its constructor, then it will not be added to the list of available modules, but its code will not be unloaded.

2.2.2 Uninstalling Modules

If a module is not needed or is acting faulty it can be removed from the system. When the module’s JAR file is removed from the modules directory, the Apache Virtual File System will send the ModuleManager a message that the file has been removed, in which case the following steps occur:

1. As of this writing there is no standardization for WebSockets in Java Servlet 3.0 so this functionality is limited to Tomcat 7+ servers that support HTTP Protocol upgrades via Apache Coyote.

7http://commons.apache.org/vfs/
2.2.3 Visiting Modules

The Module Manager encapsulates access to the various modules available to SemantEco and employs the visitor pattern to interact with the modules. The specifics of how to implement a module’s visit methods are discussed in more detail in Section 5.5. For the various core components, such as the Request object or the QueryExecutor, the Module Manager exposes a number of public methods:

- **buildUserInterface(SemantEcoUI, Request)** - When an instance of the SemantEcoUI object is instantiated (usually in response to having to render one of the tag library components, see Section 4.1.1), it is passed to the ModuleManager using this function, which in turn calls each visit(SemantEcoUI, Request) method on all of loaded modules.

- **buildOntologyModel(OntModel, Request)** - When the QueryExecutor requires the ontology model prior to executing a reasoning query, it generates an OntModel via Jena’s ModelFactory and passes it along with the active Request object to this method. The ModuleManager then calls the visit methods of all of the active modules.

- **buildDataModel(Model, Request)** - When the QueryExecutor requires a data model prior to executing a reasoning query, it generates a Model and proceeds in a similar manner as buildOntologyModel defined above. This method only calls a module’s visit method if it is an active module.

- **augmentQuery(Query, Request, Module)** - When the QueryExecutor intends to execute a query, either remotely or locally, it calls this method and passes the Query, the current Request object, and the calling Module. This method will visit each active module via its visit(Query, Request) method, assuming that module is not the same as the calling module to prevent loops where a module’s visit method may spawn further queries.

2.3 ModuleConfiguration

When each module is instantiated by the ModuleManager, the ModuleManager passes the module an instance of ModuleConfiguration via the module’s setModuleConfiguration method. Modules that wish to execute queries or retrieve runtime information about available domains, etc., should save the reference to the ModuleConfiguration object. The ModuleConfiguration object provides a number of useful methods for the module:

- **getSparqlEndpoint** - Retrieves the default SPARQL endpoint that will execute queries for this module. This can be overridden by calling a QueryFactory’s execute method that takes an endpoint URL.

- **getQueryFactory** - Obtains a QueryFactory the module can use to instantiate Query objects.

- **getQueryExecutor** - Obtains a QueryExecutor the module can use for executing queries either on remote endpoints (such as the default endpoint returned by getSparqlEndpoint) or the local model after all modules have populated it with content relevant to the actively processed Request object. The QueryExecutor object is discussed in more detail in Section 2.4.

- **getResource** - Obtains a reference to a resource in the module’s META-INF/res directory so that it can be added to the user interface.

- **generateStringResource** - Generates a resource object from a string; this is primarily used for generating dynamic content based on SPARQL queries.

- **getLogger** - Gets the default Apache log4j9 Logger for the module.

---

8 An active module is a module that does not declare a domain or has at least one domain enabled in the Domain facet

9 http://logging.apache.org/log4j/1.2/
• **getDomain** - Gets or creates a Domain object for a given URI; this should only be called by modules that also implement ProvidesDomain (note: SemantEco does not enforce this but interaction with domains are limited if a module is not also an implementation of ProvidesDomain).

• **listDomains** - Returns a java.util.List of all domains know to SemantEco at the time it is called.

### 2.4 The Query Executor

The QueryExecutor interface provides the primary method for modules to query the SemantEco knowledge base and interact with external SPARQL services via the Query object. It also provides methods for properly forming HTTP requests to external services to ease in module development by hiding many of the details related to calling remote services. Each QueryExecutor implementation must implement the following public methods:

- execute(Query) - Executes the specified query against the default endpoint and returns the response as a String.
- execute(Query, Model) - Executes the specified query against the default endpoint and loads the response into the provided Jena model. This method only is valid for Query objects that are of type CONSTRUCT or DESCRIBE and thus return an RDF graph.
- execute(String, Query) - Executes the specified query against the specified endpoint and returns the response as a String.
- execute(String, Query, Model) - Executes the specified query against the specified endpoint and loads the response into the provided Jena model. The conditions on execute(Query, Model) also apply here.
- getDefaultSparqlEndpoint() - Returns the default SPARQL endpoint. This will either be the SemantEco default or the module specific endpoint specified in the module’s properties file (see subsection 2.2.1).
- accept(String) - Adds a MIME type to the Accept HTTP header, e.g. ”application/json” or ”text/turtle”. This returns a clone of the QueryExecutor with appropriate modifications so that calls can be chained together to add multiple types and immediately perform an execution.
- executeLocalQuery(Query) - Performs a local query over the knowledge base by obtaining the combined model via the current Request object (see subsection 2.5.1).

For most modules, the general flow of information involving the QueryExecutor proceeds in the following way:

1. Construct a Query object and add appropriate triple patterns.
2. If the query is a CONSTRUCT or DESCRIBE query, construct a Jena Model object via Jena’s ModelFactory.
3. Set the appropriate accept header using accept() and then call the appropriate execute method, e.g. `config.getQueryExecutor().accept("application/json").execute(myQuery);`

Prior to execution of a query, the QueryExecutor calls the ModuleManager’s augmentQuery(Query, Request, Module) method. This causes the ModuleManager to pass each Module the query object so that they may add any additional triple patterns based on the parameters passed in the current state (encapsulated within the Request object). Modules are visited in the order that they were instantiated by the ModuleManager when the servlet was initialized. The final Module parameter is used by the ModuleManager to ensure termination by preventing an infinite loop in the event a Module creates and executes a Query from within its own visit(Query, Request) method—it is assumed that a Module has done everything it needs to do to construct the query prior to executing it.
2.5 Client-Server Communication

In this section we will discuss how SemantEco uses the @QueryMethod annotation in order to generate communication code between client and server.

2.5.1 Query Methods

To facilitate interaction between SemantEco clients and the SemantEco server, the framework provides an annotation for module designers called QueryMethod that allows the designer to identify Java methods that should be callable from client-side code. For each module with at least one QueryMethod, the framework will provide a <script> element in index.jsp under the /js/modules/ path mentioned in Section 2.1.2. The browser will attempt to load this script from this URL, which prompts SemantEco to generate the necessary glue code to support the call. This operation is done in the JavaScript generator.

The JavaScript Generator

The JavaScript generator is a utility class that is responsible for standardizing the client-server communication code to limit potential errors on the part of module designers. Each module has its QueryMethods exposed in exactly the same way and any requests to these methods from the clients are processed in exactly the same way by the SemantEco servlet. The general flow of the JavaScript generator is as follows:

1. Begin by generating an empty JavaScript object assigned to a variable with the same name as the Java class.
2. For a given module, use the Java Reflect API to obtain a list of all methods defined on the module.
3. For each method, examine its annotation set to determine if the method has the QueryMethod annotation.
4. If the method is annotated as a QueryMethod, add a key/value pair to the JavaScript object in #1 where the key is the name of the method and the value is a function that takes a dictionary of arguments, a success function, and an optional error function and performs the following actions:
   (a) Extend the SemantEco state with any optional arguments given in the arguments parameter. If an argument appears in both the state and the arguments object, the value in the arguments is used.
   (b) Using jQuery, execute an Asynchronous JavaScript and XML (AJAX) to the server at a predetermined URL in the form of /rest/<module-name>/<method-name>.
   (c) Set the success and error handlers on the AJAX request so the appropriate functions are called when the request completes.
5. If at least one method was a QueryMethod return the final object back to the servlet so it can be sent to the client, otherwise return nothing.

The Request Object

When a QueryMethod is called from a client, it is intercepted by the SemantEco servlet’s doGet method. This method identifies the request as a RESTful call to a module and proceeds to parse the parameters sent by the client into the appropriate types. This information is then encapsulated in a Request object (realized by the ClientRequest class) that can be passed to the module/method combination identified in the URL. In addition to providing modules access to the parameters of a REST call, the Request object also provides the following methods:

- getLogger - Gets an Apache log4j Logger customized for the specific request if the client sent a valid identifier for a known WebSocket outbound channel (see Section 2.6 for more details).
• **getModel** - Gets the **OntModel** containing all of the ontologies for every active module based on the **Request**’s parameters. If the **OntModel** has not yet been constructed, this will cause one to be created and the **ModuleManager** will call each active module’s **visit(OntModel)** method.

• **getDataModel** - Gets the **Model** containing all of the data for every active module based on the **Request**’s parameters. If the **Model** has not yet been constructed, this will cause one to be created and the **ModuleManager** will call each active module’s **visit(Model)** method.

• **getCombinedModel** - Gets a combined **Model** that contains the contents of both the ontology model and data model. If either model has yet to be built this will construct the missing models first following the steps above.

It is important to note that the results of each of the above methods caches its results until the processing of the **Request** completes, i.e. multiple calls to these methods will not incur any additional time penalties after the model is first built.

### 2.6 Debugging Techniques

This section discusses some of the debugging techniques that designers can use in the construction and debugging of modules.

#### 2.6.1 Web Sockets

To ease debugging server behavior during client actions, SemantEco uses the Web Socket specification to open up a persistent connection between the client and server if the client supports it. This occurs during the client’s initialization procedure. If the Web Socket initialization is successful, the SemantEco servlet stores a reference to the socket and sends back a hash of the socket’s reference that the client stores in a cookie. In future requests, the servlet will attempt to read the cookie and, if the hash identifies an open socket, will use that socket to generate a **LoggerWrapper** object that wraps the **Request**’s default logger and sends copies of all log statements to the client. Log statements received on the client via the Web Socket are printed to the client’s console. Messages may still be blocked if the log level on the server is too high for the type of log required (e.g. the developer wishes to see DEBUG messages but the server is configured for INFO and higher).

#### 2.6.2 Remote Java Debugging

In addition to using a Logger object to log events as they occur, it is also useful for designers to step through code and examine variables to identify potential logic issues and correct them. Because SemantEco uses the Maven build system it is generally the case that Eclipse does not recognize it as a servlet. Therefore, debugging the servlet is a more involved process than general servlet development done entirely in Eclipse. To debug SemantEco one should do the following:

1. Start Tomcat with `./catalina.sh jpda start` rather than `./startup.sh`. This enables the Java Platform Debugger Architecture in Tomcat’s Java Virtual Machine.
2. In Eclipse, create a new Debug configuration for the SemantEco web app.
3. Add a new **Remote Java Application** configuration.
4. Optionally, check the option that this configuration be included in the “Favorites” list.
5. Press the “Debug” button to connect to the virtual machine running Tomcat.

Now, as Tomcat executes SemantEco code it will cause Eclipse to stop at any breakpoints set by the developer.
Chapter 3

Semantic Model

In this chapter, we will present an overview of SemantEco’s ontology using OWL 2 [8]. Furthermore, we will discuss how one models regulation violations and how modules can make use of the Pellet [9] semantic web reason to write more robust queries using inference.

3.1 Sites, Measurements, and Pollution

In this section we define the Site, Measurement, and PollutedSite classes and their relationships to one another as well as external ontologies such as NASA’s SWEET ontology.

3.2 Modeling Regulation Violations

In this section we will guide the reader through examples of how to model subclasses of RegulationViolation to take advantage of the existing semantic model and queries provided by various modules (e.g. the regulation module).

3.3 Querying the Model

In this section we will instruct the reader how to make use of the semantic model to simplify queries.
Chapter 4

User Interface

In this chapter we will provide an overview of the different components in the SemantEco interface and how modules use them to provide users with tools for controlling data search.

4.1 The Semantic Entry Point: index.jsp

When a user opens SemantEco in her browser, the servlet container compiles and executes the Java Servlet Page (JSP) index.jsp, which can be found in SemantEco’s webapp directory. This file provides the primary interface for users to access data provided by the various SemantEco modules and it is automatically generated through the use of a tag library and the SemantEcoUI object constructed by the ModuleManager when it visits each of the available modules, which will be discussed in Section 4.3.

4.1.1 The SemantEco Tag Library

SemantEco provides a Java Servlet Page tag library that simplifies the generation of the main page by condensing references to the various scripts, stylesheets, and facets provided by the different modules into three separate tags: `<module:scripts />`, `<module:styles />`, and `<module:facets />`. The module namespace is explicitly defined by the Tag Library Description (TLD) WEB-INF/semanteco.tld in the web application. The implementations of the tag handlers for these tags are located in the edu.rpi.tw.escience.semanteco.taglib package. For a complete overview of JSP tag handlers see [10].

The Styles Tag

The styles tag pulls all of the resources from the SemantEcoUI object that were added via its addStylesheet(Resource) method. These are inserted by the styles tag handler as HTML `<link>` elements. It is important to note that module stylesheets are added in the order that modules are instantiated by the ModuleManager and in which the module adds them to the SemantEcoUI object. Therefore, it is advisable to tag a module’s div element with an id attribute and then structure module-specific CSS around the particular id to prevent potential conflicts with other modules.

The Scripts Tag

Like stylesheets, the scripts tag pulls all of the resources in SemantEcoUI added via its addScripts(Resource) method. It is also responsible for including references to the auto-generated JavaScript code used to provide client-server REST communication. These generated scripts are added first followed by module-specific resources, which allows the module-specific client code to manipulate the module object on the client side. For example, the RegulationModule provides QueryMethods, causing SemantEco to generate
a JavaScript object named RegulationModule on the client, and the additional behavior added by regulation.js augments the RegulationModule object with additional functions that can be called as part of the client-side application flow. This design allows module engineers to neatly collect module-specific behavior into a single client-side object.

The Facets Tag

4.2 Map Interface

In this section we will discuss how SemantEco presents known sites on a map interface.

4.3 Module Facets

Modules add facets to the SemantEco user interface via their visit(SemantEcoUI, Request) methods. In order for a Module to provide a facet to the user interface, it must store a reference to the configuration supplied by the ModuleManager via the setModuleConfiguration method. The configuration object provides two methods, getResource(String) and generateStringResource(String), that can be used to obtain Resource references to files (e.g. HTML or JSP) on disk or to programmatically constructed resources, respectively. Resources must be placed in the META-INF/res/ directory for SemantEco to locate them. To take advantage of the automatic parameter serialization mechanisms provided by SemantEco, a Module's facet must be wrapped in a <div> element whose class attribute must contain the keyword “facet”, e.g.

```html
<div class="facet">
<!-- more HTML here -->
</div>
```

Any <input> or <select> element with a facet div will be used to populate the jQuery BBQ state used to communicate application parameters between client and server. When SemantEco initializes, its client-side user interface (SemantEcoUI.js) code locates all facet elements and uses their default states to populate the jQuery BBQ state. In the event that the current URI contains state information (e.g. if the user clicked on a SemantEco URI in an email), then the data available in the jQuery BBQ state will be used to update the input elements so that they agree. SemantEcoUI also adds state change handlers to every input element so that when the user adjusts the input, the change propagates to the jQuery BBQ state.

In the event that a Module wishes to provide a facet but does not want the state information to be transmitted to the server (e.g. because the state is for client-side processing only), then the div can also be given the class “no-rest”, which informs SemantEco that the parameters related to that module should be stripped from the state prior to any server communication. This operation is accomplished by SemantEco.prepareArgs, which is called whenever the client calls one of the RESTful interfaces to the server (i.e. those methods auto-generated from QueryMethods as discussed in Section 2.5).

4.4 Hierarchical Search with JSTree

In this section we will discuss how SemantEco modules can present hierarchical search facets using OWL ontologies.

4.5 Charting with jqPlot

In this section we will discuss the use of jqPlot to provide data visualizations for the user.
Chapter 5

Writing Modules

In the previous chapters we have discussed the modular architecture that SemantEco uses for combining data, ontologies, and queries together to make a functioning system for data integration. Modules are a critical part of SemantEco and are responsible for providing functional components for the application. In this chapter, we introduce the process of constructing a new module and writing its visitor methods to provide functionality. Module authors should familiarize themselves with the Maven build system used by SemantEco for packaging its artifacts.

5.1 Creating a New Module

We provide a Maven archetype to assist module authors in the basic setup and creation of a new module. This archetype is provided both in source and via the Tetherless World maven repository. This section details how to use each system to bootstrap a module.

5.1.1 Bootstrapping from Source

Clone the SemantEco Git repository:

```
git clone https://github.com/ewpatton/SemantEco.git
```

Within the SemantEco repository is a directory called `module-archetype`. Change into that directory and build the archetype with `mvn clean install`. This will cause Maven to build the archetype and install it into the local repository.

5.1.2 Bootstrapping using Tetherless World’s Maven Repository

Note, prior to version 4.1 authors may need to amend the generated pom.xml file to include a reference to the Tetherless World Maven repository so that Maven can find the appropriate parent artifacts (see Appendix A for more details).

```
mvn archetype:generate \
-DarchetypeCatalog=https://orion.tw.rpi.edu/artifactory/libs-release-local/
```

5.2 The ProvidesDomain Interface

In this section we discuss how to declare new domains for SemantEco.

---

1See [http://maven.apache.org/](http://maven.apache.org/)
5.3 Ontology Construction
In this section we provide examples of classes that extend the SemantEco ontology as a basis for discussing how module designers can similarly extend the ontology to model new domains.

5.4 Data Conversion
In this section we provide instructions for converting data using CSV2RDF4LOD [11] and include some sample enhancements from previous conversions.

5.5 Visitor Methods
In this section we present a detailed discussion of the different visitor methods that modules must provide.

5.5.1 Adding client interfaces and behavior
In this subsection we outline how to add scripts, stylesheets, and facets to the SemantEco user interface via the UI visitor.

5.5.2 Populating the Data Model
In this subsection we demonstrate how to read data from an external triple store into the local model so that it will be exposed to the other modules and available for any local queries.

5.5.3 Extending the Ontology Model
In this subsection we demonstrate how to extend the ontology model by reading in external ontologies using Jena [12].

5.5.4 Augmenting Queries from other Modules
In this subsection we discuss how to augment queries and use the metaquery methods exposed by the Query object to make decisions about what modifications, if any, should be performed.

5.6 Exceptional Behavior
In this section we discuss alternative behaviors for loading and querying data and the reasons why one might want to use them over the visitor methods outlined in the previous section.
Chapter 6

Available Modules

In this chapter we will provide an in-depth look at the current set of modules for SemantEco and the different datasets and query methods they provide. We will also discuss the different classifications of modules (i.e. core, data provider, and query-altering).

6.1 Core Modules

This section will discuss the core modules that are required in the general flow of SemantEco. Removing any of these modules will break the user interface as it is currently engineered (although in many cases an alternative module could be supplied).

6.1.1 Domain Module

In this subsection we discuss the domain module’s purpose.

6.1.2 Data Source Module

In this subsection we discuss the data source module’s purpose.

6.1.3 Regulation Module

In this subsection we discuss the regulation module’s purpose and query methods.

6.1.4 Entity Type Module

In this subsection we discuss the entity type module’s purpose.

6.1.5 Zip Code Module

In this subsection we discuss the zip code module’s purpose and query methods.

6.1.6 Debugger Module

In this subsection we discuss the debugger module and how module designers can use it to assist in debugging modules.
6.2 Data Provider Modules

In this section we discuss data provider modules and how they provide data and ontologies to SemantEco through the various visitor methods discussed in Section 5.5.

6.2.1 Water Data Provider

In this subsection we discuss the water data provider, its ontologies, and its data sources.

6.2.2 Air Data Provider

In this subsection we discuss the air data provider, its ontologies, and its data sources.

6.2.3 Species Data Provider

In this subsection we discuss the species data provider, its ontologies, and its data sources. We also discuss its use of the hierarchical search facet to assist users in locating data.

6.3 Query-modifying Modules

In this section we discuss modules that do not provide additional data, but instead make use of the query visitor pattern to augment queries from other modules to restrict them to certain criteria.

6.3.1 Time Module

In this subsection we discuss the time module and how it modifies queries pertaining to measurements to limit them to certain time frames.

6.3.2 Industry Module

In this subsection we discuss the industry module and how it uses North American Industry Classification System (NAICS) to restrict queries regarding facilities to those facilities in a particular industry.

6.3.3 Characteristic Module

In this subsection we discuss the characteristic module and its ontology that merges ChEBI and SNOMED. Furthermore, we discuss the use of this ontology in a hierarchical search facet.
Chapter 7

Conclusions

In this chapter we will present our conclusions.
Appendix A

Maven Build System

SemantEco uses the Apache Maven build system to handle its build process. Each component of SemantEco is a module in a top-level Maven POM file. Development of stand-alone modules can reference the build artifacts via Tetherless World’s Artifactory installation. To do so, one must put the following snippet in their module’s pom.xml:

```xml
<repositories>
  <repository>
    <id>tetherless</id>
    <url>https://orion.tw.rpi.edu/artifactory/libs-release</url>
  </repository>
</repositories>
```

Note: This step is not required if you have added your module to the existing SemantEco repository as this step has already been done in the root pom.xml.

A.1 Build-time Profiles and Application Configuration

SemantEco has two Maven profiles that control its build process. The default profile is for development purposes and is configured to deploy the SemantEco WAR file to a local servlet container instance. The development profile also includes the Debugger module that provides an alternate interface for testing SPARQL queries against Pellet-backed data models. The second profile is for production deployment. It is configured specifically for SemantEco deployment at the Tetherless World Constellation, so its parameters should be adjusted according to the environment where it is set to be deployed. To build a production version, one must tell Maven to activate it via the `-P` command line parameter (i.e. `mvn clean install -P production`).

Additionally, SemantEco has a number of properties that govern its behavior at run time. It reads these properties from a file called semanteco.properties. These properties come from the `<properties>` element in the webapp’s pom.xml file. As of this writing, the current set of properties are:

- **debug** This is set by development mode and increases the amount of logging performed by the application for debugging purposes.

- **triple-store** The default triple store to execute queries against. In development it points to [http://was.tw.rpi.edu/virtuoso/sparql](http://was.tw.rpi.edu/virtuoso/sparql) and in production points to [http://localhost:3003/](http://localhost:3003/).

- **baseUrl** The base URL is the user-facing path for SemantEco. It uses this to construct its RESTful URLs for client-server communication. This should not be present in development. The default for production is [http://aquarius.tw.rpi.edu/projects/semanteco/](http://aquarius.tw.rpi.edu/projects/semanteco/).

In addition to these properties, SemantEco also consumes a single environment variable, `semantEcoRootPath`, that defaults to its location within the servlet container’s webapps directory. This is used for SemantEco to generate logs, and changing it will change where the log files are stored.
Appendix B

Source Code Management

The original line of SemantEco software was developed and maintained at Google Code[1]. Due to the success of the project and the large number of exploratory offshoots, it was necessary to move to a faster source code management system. To this end, SemantEco has been migrated to use the Git source code management tool to maintain its source history. The project is hosted via Github at [https://github.com/ewpatton/SemantEco](https://github.com/ewpatton/SemantEco) and maintains the entire history of the project, including the original Subversion repository from Google Code.

Parties interested in submitting code to SemantEco should use the Github application to fork the repository and submit pull requests back to the main repository. The main repository uses the Git Flow[2] development paradigm, so pull requests should point to a branch from `develop` that contains the necessary code for the feature. Multiple features should be submitted on multiple branches. Experts may feel free to use command-line Git but are discouraged from rebasing unless absolutely necessary. Any pull requests that would cause conflicts with `develop` or fail unit tests will be rejected.

References


