Researcher Support System (RSS)
Software Architecture Requirements and Design Specifications
(version 0.1)

Fritz Solms, Vreda Pieterse, Stacey Omeleze & Antonia Michael
Dept Computer Science, University of Pretoria

March 15, 2016

Contents

1 Software architecture overview

Figure 1 shows a high-level overview of the software architecture. In particular, it shows the decomposition of the system into layers with abstract responsibilities, the core architectural components of the system and the concrete frameworks to be used when realizing these architectural components.

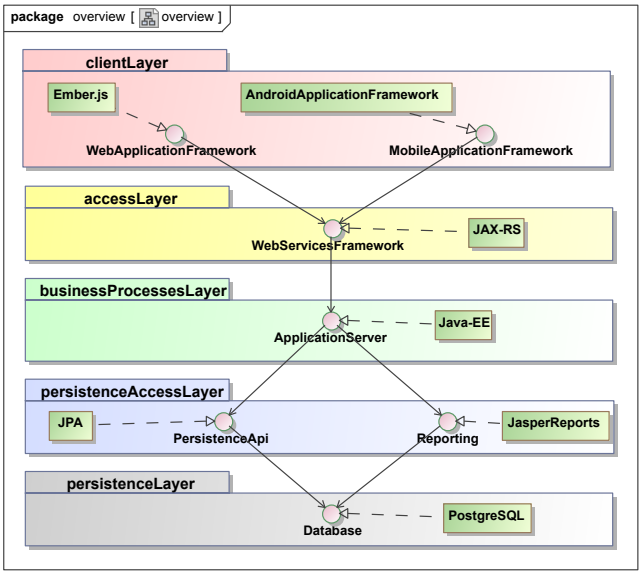

Figure 1: A high-level overview of the software architecture of the Research Support System.

2 Overall software architecture

This section specifies the software architecture requirements and the software architecture design for the first level of granularity – the system as a whole. The output will be the high level software architecture components, the infrastructure between them and the tactics used at the first level of granularity to realize the quality requirements for the system.

Subsequent sections will focus on the software architecture requirements and design for these high-level architectural components. Note that many of the architectural requirements (particularly the quality requirements) will be propagated down to lower level architectural components.

2.1 Architecture requirements

2.1.1 Access and integration requirements

This section discusses the software architecture requirements — that is the requirements around the software infrastructure within which the application functionality is to be developed. The purpose

of this infrastructure is to address the non-functional requirements. In particular, the architecture requirements specify

- the architectural responsibilities which need to be addressed,
- the access and integration requirements for the system,
- the quality requirements, and
- the architecture constraints specified by the client.

2.1.2 Quality requirements

The quality requirements are the requirements around the quality attributes of the systems and the services it provides. This includes requirements like maintainability/flexibility, extensibility, performance, scalability, security, auditabilty, usability, and testability requirements.

2.1.2.1 Flexibility It is important that the system architecture is such that one can easily add different access channels to the system. This is particularly important in the context of the proliferation of connected devices in a world transiting to fully embrace the Internet of Things (IoT).

Furthermore, persistence architectures and reporting infrastructures are rapidly evolving as can be seen from the rapid growth of NoSQL databases, semantic knowledge repositories and big data stores. In this context it is important that the application functionality is not locked into any specific persistence technology and that one is able to easily modify the persistence provider and reporting framework.

2.1.2.2 Maintainability Amongst the most important quality requirements for the system is *maintainability*. It should be easy to maintain the system in the future. To this end

- future developers should be able to easily understand the system,
- the technologies chosen for the system an be reasonably expected to be available for a long time,
- and developers should be able to easily and relatively quickly
  - change aspects of the functionality the system provides, and
  - add new functionality to the system.

In addition, it should be very for TechTeam members to modify reports and to add new reports to the system.

2.1.2.3 Scalability

1. Initially the system must support the ability to host the research community of the size of the Computer Science department of the University of Pretoria, i.e. supporting a maximum of 50 concurrent users.
2. The architecture of the system should, however, be such that it can be easily scaled to research communities of several hundred researchers.

2.1.2.4 Performance requirements The system does not have particularly stringent performance requirements.

1. All non-reporting operations should respond within less than 0.2 seconds.
2. Report queries should be processed in no more than 5 seconds.

The above figures do not include the network round-trip which is outside the control of the system.

2.1.2.5 Reliability The system should provide by default a reasonable level of availability and reliability and should be deployable within configurations which provide a high level availability, supporting

- fail-over safety of all components and
- a deployment without single points of failure.

Hot deployment of new/changing functionality is not required for this system.

2.1.2.6 Security Initially the system needs to support only

- authentication against a chosen user repository (initially against the LDAP repository of the Computer Science Department of the University of Pretoria),
- and a flexible, configurable authorization framework allowing administrators to configure the access to any particular service based on the user role and other factors like the user status.

In future the system is expected to also enforce confidentiality through encrypted communication and protection against man-in-the-middle attacks through hashing.

2.1.2.7 Auditability The system will log all requests and all responses (including exceptions) for all user services provided by the system.
For each request the log should contain an entry with

- an id for the log entry,
- the userId of the user requesting the service,
- the date/time stamp when the request was made,
- the user service requested, and
- the request object stringified as JSON with any sensitive information like passwords removed.

For each response the log should contain an entry with

- an id for the log entry,
- the id of the corresponding request entry,
- the date/time stamp when the response is provided, and

- the response object stringified as JSON with any sensitive information like passwords removed.

The system will provide only services to extract information from the audit log and will not allow the audit log to be modified.

Audit logs will be directly accessible to both, humans and systems.

2.1.2.8 Testability All services offered by the system must be testable through

1. automated unit tests testing components in isolation using mock objects, and
2. automated integration tests where components are integrated within the actual environment. which test.

In either case, these functional tests should verify that

- that the service is provided if all pre-conditions are met (i.e. that no exception is raised except if one of the pre-conditions for the service is not met), and
- that all post-conditions hold true once the service has been provided.

In addition to functional testing, the quality requirements like scalability, usability, auditability, performance and so on should also be tested.

2.1.2.9 Usability Usability is one of the most important quality attributes. The system should be intuitive and efficient to use. Computer literacy can be assumed, but the time it takes users to add or modify a paper should be kept to a minimum. Error messages should be self-explanatory and as much as possible of the input validation should be done in the client.

- users with a literacy computer rate of an average first year students are able to use the system without any guidance or documentation,
- whether users find any aspects of the system cumbersome, non-intuitive or frustrating.

In addition the system should be usable by users which have different language preferences. Initially the system needs to support the three official languages of the University of Pretoria, Afrikaans, Sepedi and English.

Innovation and novel approaches are encouraged for this system. Nevertheless, in cases where a UI design aspect deviates from accepted standards and norms, there should be a strong usability reason for having deviated from these norms.

2.1.2.10 Integrability The system should be able to easily address future integration requirements by providing access to its services using widely adopted public standards. All use case which are available to human users should also be accessible from external systems.

2.1.2.11 Deployability The system must be buildable from source and build scripts only.

The system must be deployable

- on Linux servers,
- in environments using different databases for persistence.

The system should ultimately be packaged as a Docker image which is deployable on a Docker container installed on a virtual or physical Linux server.

2.1.3 Architectural responsibilities

The architectural responsibilities for the system as a whole are shown in Figure **??**.

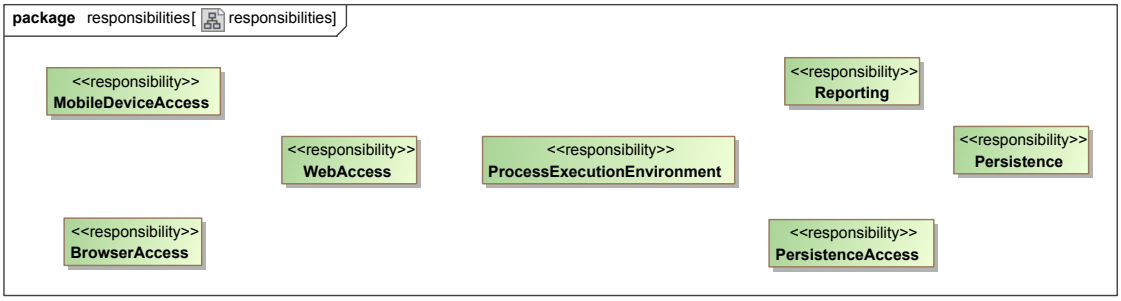

Figure 2:

The system requires an environment within which the business processes realizing the services are executed. These services need to be made available to humans and systems over the web. Humans need to access the services via mobile devices and web browsers. The business processes require access to a persistence provider (a database) off which configurable reporting can be done.

2.1.4 Architecture constraints

The choice of architecture is largely unconstrained and the development team has the freedom to choose the architecture and technologies best suited to fulfil the non-functional requirements for the system subject to

1. the architecture being deployable on Linux servers, and
2. the architecture using only open source frameworks and tools.

2.2 Architecture design

This section specifies the very high-level software architecture design, i.e. the software architecture design for the first level of granularity. It includes the allocation of architectural responsibilities to architectural components, any tactics which should be used at the current level of granularity to address quality requirements,

2.2.1 Architectural tactics

At this high level of abstraction, we do not specify any architectural tactics in order to concretely address the quality requirements for the system.

2.2.2 Architectural components addressing architectural responsibilities

Figure 3 shows the allocation of architectural responsibilities to architectural components.

As can be seen from the diagram,

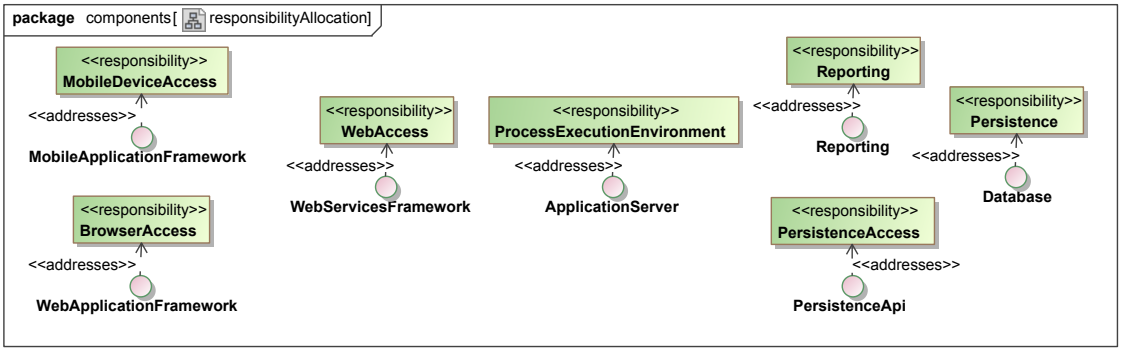

Figure 3: The abstract architectural components to which the architectural responsibilities are assigned.

- the responsibility of providing an execution environment for business processes is assigned to an application server,
- the responsibility of persisting domain objects to a database,
- the responsibility of providing access to the persistence provider to a persistence API,
- that of providing an environment for specifying and executing reports to a reporting framework,
- that of providing human access channels and external systems web access to the system services to a web services framework,
- that of providing users system access from mobile devices to a mobile application framework, and
- that of providing users access through their browser to a web application framework.

Here

2.2.3 Infrastructure

Architectural (structural) patterns are used to constrain the infrastructure between the architectural components. Amongst the most important architectural requirements are

- to provide different access channels to the system with the different access channels potentially changing over time,
- and to be flexible around the persistence and reporting infrastructures.

The *layered architectural pattern* provides an infrastructure which limits access of component within one layer to components which are either in the same or the next lower level layer. One of its strengths is that a can be relatively easily replaced. In particular, one can add further access

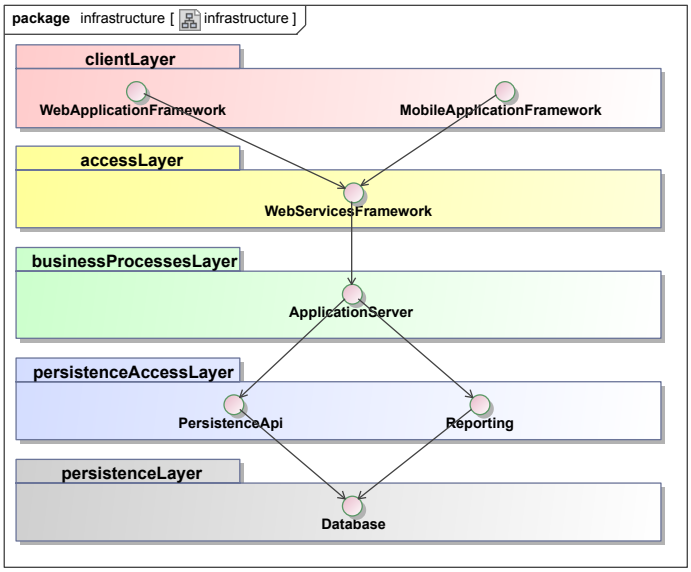

Figure 4:

channels without changing any lower layers within the software system and changing the persistence provider would only require changing the persistence API and possibly the reporting framework.

Figure 4 depicts the layered infrastructure between the high-level architectural components.

2.2.4 Concepts and constraints for application components

There are no application components which will be deployed at the first level of granularity. The application components will be hosted within the architectural components (e.g. the process execution environment, the mobile application, the database and so on).

3 Application server

The application server is the architectural component within which business processes are to be deployed and executed. It is the component hosting the business processes layer of the application.

3.1 Software architecture requirements

The architectural requirements for the application server include the refined quality requirements and the architectural responsibilities. The architectural constraints for this lower level component are the same as for the system as a whole.

3.1.1 Quality requirements

A large subset of the quality requirements for the system as a whole need to be propagated to this lower level component, i.e. most of them are also quality requirements for the application server itself.

In particular, the scalability, maintainability, auditability and deployability requirements specified for the system as a whole are directly applicable for the application server. However, some of the higher level quality requirements need to be refined for this lower level component.

3.1.1.1 Flexibility Flexibility requirements for the application server include

- the ability to deploy versions of the system with minimum system down-time, and
- the ability to easily replace one lower level service provider with another.

3.1.1.2 Reliability At this level of the business processes layer we require reliability around the business processes themselves. In particular, we require that service requests are either realized as per services contract and that business processes for these services are not partially executed, i.e. that only a subset of the post-conditions are fulfilled. Furthermore, in cases where the service is not provided the reason for refusing the service must be communicated to the calling entity.

3.1.1.3 Security The security requirements which are relevant for the business processes layer revolve around authorization. The application server used must support role-based authorization. No direct access should given to the database, i.e. data is only accessible through the services offered by the system, with the application server authenticating itself with the database.

3.1.1.4 Testability The application server framework chosen must support out-of-container testing. As a nice-to-have it could also support embedded container unit testing.

Hot deployment
Scalability
Security (authorization)
Scheduling
CPU access (Thread provision)
Auditability

3.1.2 Architectural responsibilities

The architectural responsibilities of the application server are shown in Figure 5.

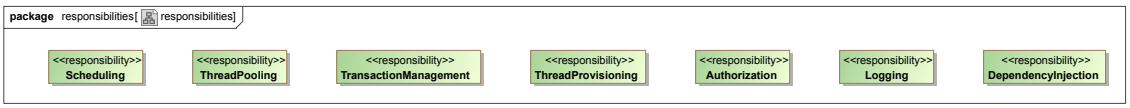

Figure 5: The architectural responsibilities of the application server

3.2 Architecture design

3.2.1 Tactics

3.2.1.1 Flexibility tactics Tactics which should be used by any chosen application server framework to concretely address the flexibility requirements application server should support

- hot-deployment, i.e. the ability to deploy new versions of the system into the live environment without bringing down the application, and
- component (contract) based software development with dependency injection.

3.2.1.2 Maintainability tactics The application server should be based on a public standard with multiple implementations including at least one open-source implementation of the standard being available.

3.2.1.3 Reliability tactics The application server must provide support for transactions which may enlist a variety of resources. In particular, the system requires the ability to commit and roll-back across both, the persistence provider (i.e. the database) as well as the mail server adapter.

The transaction support should be declarative, i.e. that one specifies the transactional requirements and does not hard code transaction boundaries. This has the advantage that services can be reused across different business processes requiring different transactional boundaries. For example, the `sendMessage` service ,may, at times, run within its own transaction (with transaction boundaries at the beginning and end of that service), whilst the service can be reused within the `changePublicationState` where the transaction is only committed at the end of this higher level service committing across the database as well as the mail server adapter (and not after sending each individual activity notification message to each of the authors who requested such notifications).

A nice-to-have is the ability to deploy the application server in a clustered configuration over multiple physical or virtual servers in order to have fail over safety (as well as increased scalability).

3.2.1.4 Security The security requirements for the business processes layer is *role based authorization*. The framework(s) used must support

- declarative role based authorization of services (i.e. that one can specify for a service which security roles a user must have in order to be able to make use of the service), as well as
- more sophisticated authorization rules which may depend on the details of the request object and the current state of the application.

3.2.1.5 Auditability tactics Auditability is an important quality requirement and needs to be addressed in the business processes layer. The auditability specification must be maintainable. To this end, the application server must support specifying logging logic (advices) as cross-cutting concerns within aspects or at least provide the ability to apply logging via interception.

3.2.1.6 Testability tactics In order to be able to reuse unit tests as integration and regression tests the frameworks and technologies used at the business process layer must support dependency injection. The application server must support hosting code which can be tested outside the application server.

A nice-to-have is the support for in-application-server testing.

3.2.2 Architectural components

The architectural components of the application server are shown in Figure 6.

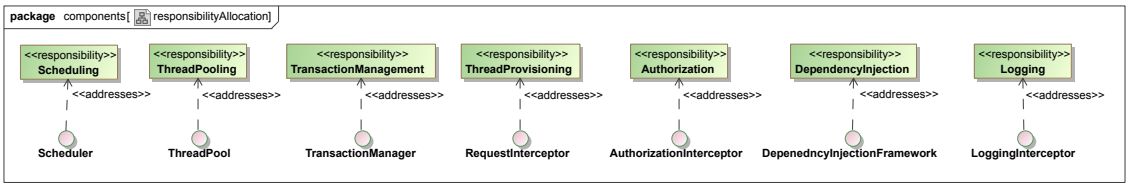

Figure 6: The abstract components to which the architectural responsibilities are assigned.

3.2.3 applicationComponentConceptsAndConstraints

The central concept which will be used to specify application logic (the business processes) are the concepts of

- a *service contracts* which encodes the requirements for a service and
- a *service* which encodes the concrete implementation of a service.

These services which encapsulate the processes will be constrained to be *stateless*, i.e. no state is maintained across service requests[1].

Long living state is maintained in domain objects which are typically persisted and which should not have any business logic.

3.2.4 Frameworks and technologies

Ember.js MVC based open source web application framework - Solid framework with a well defined structure for web applications which assists maintainability and flexibility, i.e. the framework forces a consistent, maintainable structure on a web application and different development teams can easily work on each other's code (particularly suited to long-lived applications) - solid documentation and clear guidance on how to use the framework - very efficient and faster than Angular.js - provides Module to work with data which integrates seamlessly with JSON - eliminates boiler plate code and uses naming convention over configuration - particularly aligned with the spirit of the web and suited for RESTful applications - two-way data binding - uses Handlebars template engine (very powerful and efficient, but not DOM aware)

Constraints: - do not use ORM to directly access database - go through REST layer

Other contenders: - Angular.js * powerful, flexible framework with good modularization and extensive module repositories * higher risk for creating a messy, web app which is difficult to maintain * uncertainty around the massive changes from version 1 to version 2 * inefficient two-way data binding

[1]Note that a service may maintain state for the duration of processing a particular service request, but that no state should be maintained from the processing of one request to the next.

- Backbone * very minimalistic framework * does not include template engine but can be integrated with several ones * unstructures and leaves a lot of design decisions to developer teams which may result in messy, difficult to maintain code. * no support for two-way binding * direct DOM manipulation making it difficult to test and to reuse

Build tool: Broccoli (integrates well with npm repositories) - very fast incremental build system which only rebuilds changed files - uses JavaScript and hence not another syntax/language to learn (maintainability) - supports rebuilds on file changes

QUnit for unit testing

3.2.4.1 Concrete realization of architectural components The Java-EE reference architecture addresses largely all the architectural responsibilities assigned to the application server. The concrete components addressing the required responsibilities are shown in Figure 7.

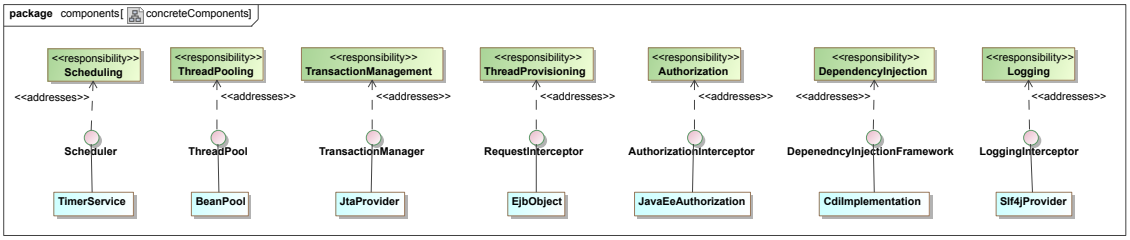

Figure 7: The components in Java-EE addressing the architectural responsibilities of the application server

3.2.4.2 tactics With the exception of full aspect-oriented programming, Java-EE-7 supports all architectural tactics specified in the software architecture design for the application server. One can use Aspect-J with compile time weaving within Java-EE, but the global interceptors provided within Java-EE provide a simpler, yet sufficient solution for specifying logging logic.

3.2.4.2.1 Concrete realization of flexibility tactics within Java-EE Java-EE supports

- Java-EE is a component-based framework requiring decoupling through interfaces/contracts.
- CDI (Context and Dependency Injection) is fully supported within Java-EE with dependency injection requested via `@Inject` annotations.
- Hot-deployment is supported by most application servers including the most widely used open-source application servers like *JBoss* and *Glassfish*.

3.2.4.2.2 Concrete realization of maintainability tactics within Java-EE Java-EE is based on a public standard which is widely supported and hence the support for this technology is secured for the foreseeable future. The technology puts a rigorous framework in place for defining services within stateless session beans which are decoupled through interfaces/contracts. Maintainability is enhanced by dependency injection which allows for changing service providers at

any level of granularity without making code changes. Furthermore the replacement of such new or improved functionality can be done within a live system using hot deployment.

3.2.4.2.3 Concrete realization of scalability tactics The research support system does not have particularly stringent scalability requirements. Nevertheless, at times the system can be used by at least many tens of concurrent users. Java-EE uses thread-pooling, object-pooling, connection-pooling and clustering to improve scalability.

3.2.4.2.4 Concrete realization of reliability tactics within Java-EE Java-EE supports container-managed transactions across multiple transaction-supporting resources with two-phase-commit. Transaction requirements can be specified declaratively. In particular, annotating services (methods) with `TransactionAttribute.Required` specifies that the service must run under transactional control and that any resources used within the service will be enlisted within the transaction for atomic commits and roll-backs. If that service is called outside a transaction scope, a new transaction will be created when the service is called and the transaction is committed on normal termination of the service and rolled back if an exception is raised. Any lower level services called from within such a service which are annotated with the same transaction attribute will be embraced within the transactional scope of the outer service. Similarly, if this service is called from a higher level service which is already under transactional scope, it will be executed within the transaction of the higher level service.

In addition to transactions support, Java-EE application servers also commonly support deployment within a clustered environment to further improve reliability (and scalability).

3.2.4.2.5 Concrete support for security tactics Java-EE supports role-based authorization in the form of both, container managed declarative authorization and programmatic authorization for more complex authorization logic. In its simplest form, methods are simply annotated with the required security roles, i.e. which security role a user must have been assigned to in order to make use of the service.

3.2.4.2.6 Realization of auditability tactics within Java-EE Java-EE does not have explicit support for auditability, but it does allow for intercepting all services with a global interceptor which can be used to log service requests and results including the information about the calling entity (user principal and roles).

3.2.4.2.7 Support for testability tactics Java-EE has good support for dependency injection. Furthermore, the architecture (e.g. the fact that the application code is executing within a Java-EE application server) is not hard-coded within the application logic, but any Java-EE specific information is specified via annotations. All application code can thus be executed outside a Java-EE container (e.g. for unit, integration and regression testing). Dependency injection can be used to inject wither mock or actual dependencies allowing for a unit test to be reused for integration and regression testing.

Finally, Java-EE-6 onwards includes the requirement for embedded containers. This does allow for bringing up a Java-EE container (as well as an embedded (in-memory) database) within the run-time environment of the unit test.

3.2.4.2.8 Deployability of Java-EE application servers Java-EE application servers are themselves Java applications which can be run in any compliant Java run-time environment and which are thus platform and operating system independent. Java-EE application servers are commonly executed on Linux servers.

3.2.4.3 Application component concepts and constraints Within the Java-EE reference architecture,

- services contracts are represented by *Java interfaces*,
- stateless services are realized within methods of *stateless session beans*, and
- domain objects are realized as *Java entities*.

Note that stateful session beans and message driven beans should not be used for the research support system.

Lower level services should be always decoupled from higher level services via interfaces/contracts and dependency injection.

4 Database

The system does not have particularly stringent scalability requirements and it is most important that a database techniology is chosen for which there are mature reporting frameworks available. For this reason the system will use a relational database.

The entire code of the system should be decoupled from which concrete database is used. For the mini-project *ProstgreSQL* will be used. It is a mature, efficient and reliable relational database implementation which is available across platforms and for which there is a large and very competent support community.

5 Persistence API

The persistence API provides abstracted access to a persistence to a persistence provider (database) whilst remaining decoupled from the database technology as well as the concrete database selected for the system. It also implements a range of tactics in order to concretely address quality requirements required from the persistence domain.

5.1 Software architecture requirements

The architectural requirements for the persistence API include the refined quality requirements and the architectural responsibilities. The architectural constraints for this lower level component are the same as for the system as a whole.

5.1.1 Quality requirements

Particularly, scalability, reliability, flexibility and maintainability are important for the persistence API.

Hot deployment

Scalability
Security (authorization)
Scheduling
CPU access (Thread provision)
Auditability

5.1.2 Architectural responsibilities

The architectural responsibilities of the persistence API are shown in Figure 8.

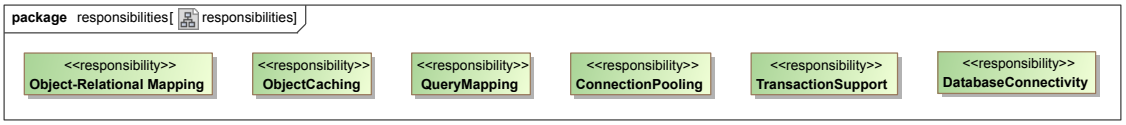

Figure 8: The architectural responsibilities of the persistence API

5.2 Software architecture design for the persistence API

5.2.1 Architectural tactics

The persistence API should use

- *object-relational mapping* to reduce code bulk, improve maintainability and allow for decoupling from the persistence provider
- *query mapping* from queries across a graph of Java objects onto the database queries used in the selected database technology and provider,
- *object caching* to improve scalability and performance,
- *transactions* with 2-phase commit to improve reliability of processes, and *connection pooling* to improve performance and scalability.

5.2.2 Architectural components

The architectural components of the persistence API are shown in Figure 9.

5.2.3 Application component concepts and constraints

Within the persistence domain the application concepts (i.e. concepts within which the application functionality is specified) are

- *domain objects* (entities) which host long-living state, and
- *queries across object graph of domain objects* through which the state information is retrieved and through which the state of the domain objects is modified.

These domain objects are constrained to be devoid of any business processes - they purely hold data.

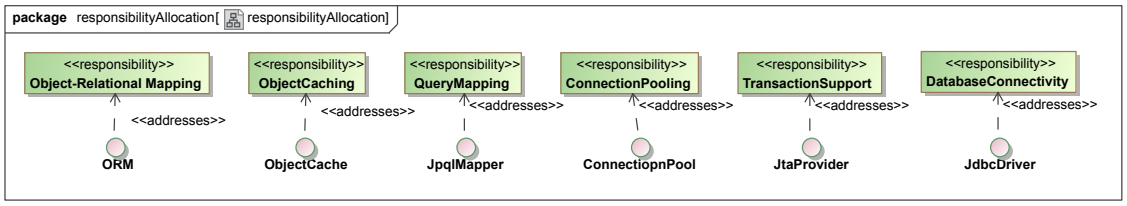

Figure 9: The abstract components to which the architectural responsibilities are assigned.

5.2.4 Frameworks and technologies

A JPA (Java Persistence API) provider will be used as a persistence API. The default concrete implementation packaged with the chosen application server (*EclipseLink* in the case of *Glassfish*) will be used. The persistence context (EntityManager) will be dependency injected into services requiring access to persistent data. JPA provider do implement

- *object-relational mapping* including mapping of relationships between objects via a provided ORM,
- *query mapping* from object-oriented queries across the domain objects graph to queries for a specific database provider (e.g. ontio SQL),
- *object caching* (within the persistence context),
- *transaction support* though the *Java Transation API*, and
- *connection pooling* through a JCA connector based implementation of a JDBC driver.

Queries will be specified as JPQL (Java Persistence Query Language) queries which are queries across Java entities (domain objects).

6 Web services framework

The web services framework is a wrapping layer of the services (business processes) layer making the services available in a technology-neutral way over the Internet, i.e. such that the services can be accessed from a wide range of technologies through the Internet. This decouples the user-interface technologies from the back-end technologies, allowing either to vary independently.

6.1 Architecture requirements

The web services framework should be based on a public standard and open-source implementation should be available. The quality requirements for this layer is mainly that of *maintainability* (this is a plumbing layer and its code should be largely auto-generated) and *integrability*.

6.2 Architecture design

For the web services framework the project will use the *Jersey* implementation of the *JAX-RS* standard shipped with the Java-EE application server. Marshalling and de-marshalling of Java data objects onto/from JSON will be done using the *MOXy* implementation of the *JAXB* standard. This is incorporated within Java-EE 7. Note that the marshalling and de-marshalling is done automatically when annotating the services as JSON producing and consuming services

```
1
2  @Path("publication")
3  class PublicationResource
4  {
5
6      @PUT // adding a resource
7      @Consumes(MediaType.APPLICATION_JSON)
8      public void create(Publication publication)
9      {
10       publications.addPublication(new AddPublicationRequest(publication))
11         ...
12     }
13
14     @GET // retrieving a resource (not changing anything in the system
15     @Path("{id}")
16     @Produces(MediaType.APPLICATION_JSON)
17     public Publication get(@PathParam("id") String id)
18     {
19        ...
20     }
21
22     @POST // modifying a resource
23     @Consumes(MediaType.APPLICATION_JSON)
24     public void update(Publication publication) throws ...
25     {
26       publications.changePublicationState(new ChangePublicationStateRequest(publication))
27       {
28         ...
29       }
30     }
31
32     @Inject
33     Publications publications;
34 }
35
36 public interface Publications // the contract
37 {
38     public AddPublicationResponse addPublication(AddPublicationRequest request) thows ...
39
40     ...
41 }
42
43 @Stateless
44 @Local
45 public class PublicationsImpl implements Publications
46 {
47     public AddPublicationResponse addPublication(AddPublicationRequest request) thows ...
48     {
49       ...
50       entityManager.persist(publication);
51       ...
52     }
53
54     @PersistenceContext
55     private EntityManager entityManager;
56 }
57
```

```
58 @Stateful
59 @Path("publications")
60 public class PublicationsImpl implements Publications
61 {
62     public Publication getPublication(
63 
64     @POST
65     @Produces(MediaType.APPLICATION_JSON)
66     @Consumes(MediaType.APPLICATION_JSON)
67     public MyObject echoObject(MyObject mo) {
68         return mo;
69     }
70 }
```

7 Web application framework

This section specifies the software architecture requirements and the software architecture design for web application framework.

7.1 Architecture requirements

The web application framework provides the software architecture for the software providing browser based access to human users.

7.1.1 Access and integration requirements

The web application must be usable by humans using any standards compliant web browser. The system should be usable by the latest versions of the Mozilla Firefox, QupZilla, Midori, Chrome, Safari, Opera and Internet Explorer.

The web application will need to integrate with the services/business processes layer via a REST API.

7.1.2 Quality requirements

The most important quality requirements for the web application layer are

- usability,
- maintainability,
- flexibility, and
- performance.

7.1.3 Architectural responsibilities

The architectural responsibilities of the web application framework are shown in Figure **??**.

7.1.4 Architecture constraints

The architectural constraints of the application as a whole are propagated directly to all components including the web application framework. It must hence be open source and make use of public standards.

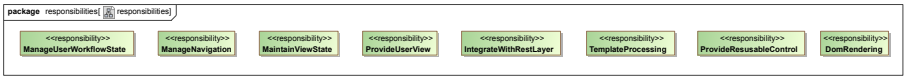

Figure 10: The architectural responsibilities of the web application framework

7.2 Architecture design

This section specifies the very high-level software architecture design, i.e. the software architecture design for the first level of granularity. It includes the allocation of architectural responsibilities to architectural components, any tactics which should be used at the current level of granularity to address quality requirements,

7.2.1 Tactics

Tactics which should be used to address the quality requirements should include

- *caching* of pre-generated and pre-populated HTML pages for performance
- *templating* for development ease, maintainability and flexibility,
- *pre-compilation* of templates,
- *automatic model population* from REST-JSON for development ease, maintainability and flexibility,
- *virtual DOM* for in-memory updates, incremental builds and efficient diffing based on differentiation between static and dynamic DOM elements (with only the latter having to be checked for updates).

7.2.2 Architectural components

The architectural components of the web application framework are shown in Figure 11.

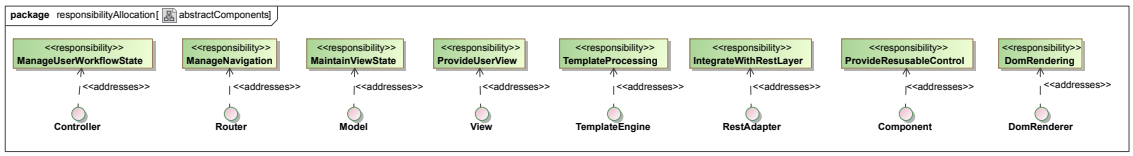

Figure 11: The abstract components to which the architectural responsibilities are assigned.

7.2.3 Frameworks and technologies

The web application architecture will be based on *Ember.js*. *Ember.js* is a MVC based open source web application framework which uses the *Handlebars template engine* which is very powerful and efficient, but not currently yet DOM aware – future versions will be DOM-aware.

Core reasons for using *Ember.js* include

- It is a framework which enforces defined structure on web applications. This assists maintainability and flexibility, enabling different development teams can easily work on each other's code. This is particularly important as one can expect the application to be long-lived, requiring maintenance by generations of TechTeam members.
- Maintainability is further improved by extensive documentation and clear guidance on how to use the framework.
- The framework is very efficient and faster than many others (e.g. *Angular.js*).
- Ember's `Model` supports seamless integration with REST and JSON.
- The framework eliminates most boiler plate code and uses naming convention over configuration. This reduces development time and maintenance cost and improves flexibility.
- The framework is cleanly developed within the spirit of the web (e.g. the consistent URL handling) and is well suited as a front-end for RESTful application wrappers.
- Two-way data binding improves responsiveness and usability.

Other web application frameworks which were considered include

Angular.js which is a powerful flexible framework with good modularization and extensive module repositories. With *Angular.js* there is, however, a higher risk for the application to degenerate into an inconsistent and difficult to maintain web application as it does not enforce a consistent way of developing the application. Furthermore, significant API changes between recent versions has had a negative impact on code stability and maintainability. The two-way binding and the updating of athe virtual DOM is also less efficient in *Angular.js*.

Backbone which is a very minimalistic framework. It has the advantage of being easier to understand, but because it is so minimalistic, it requires more code as well as development and maintenance of some boiler-plate code. The framework does not enforce an application structure and a lot of design decisions to developer teams which may result in messy, difficult to maintain code. Applications typically bind directly to the DOM resulting in code which is difficult to maintain and to reuse and there is no support for two-way binding.

7.2.3.1 Concrete realization of architectural components within Ember.js The concrete components addressing the required responsibilities are shown in Figure 12.

7.2.3.2 Tactics Tactics which should be used to address the quality requirements should include

- *caching* of pre-generated and pre-populated HTML pages for performance
- *templating* for development ease, maintainability and flexibility,
- *pre-compilation* of templates,
- *automatic model population* from REST-JSON for development ease, maintainability and flexibility,

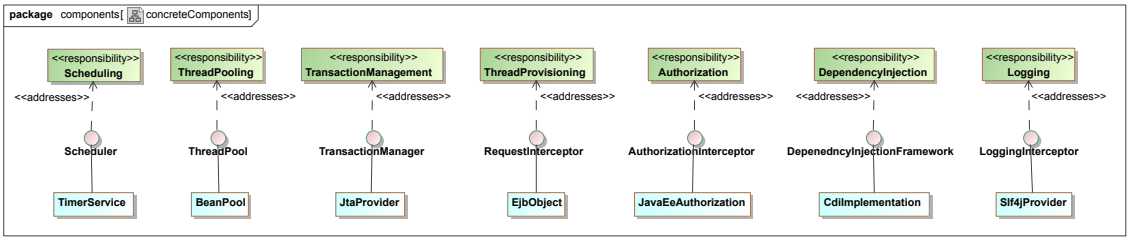

Figure 12: The components within Ember.js addressing the architectural responsibilities of the application server

- *virtual DOM* for in-memory updates, incremental builds and efficient diffing based on differentiation between static and dynamic DOM elements (with only the latter having to be checked for updates).

7.2.3.3 Tools The build tool which should be used for the web application is *Broccoli*. It

- integrates well with *npm*,
- allows for very fast incremental build system which only rebuilds changed files,
- uses JavaScript and hence there is no need to learn another syntax/language – this assists maintainability
- and support automatic rebuilds on file changes.

For testing of the web application, *QUint* should be used.

7.2.4 Concepts and constraints for application components

In Ember.js, application development is guided by the *Model-View-Controller* (MVC) pattern.
The application concepts used in Ember.js include

Model which maintains the data for the views,

Views generated from templates presenting the visual representation of the user interface.

Controllers which manage the user workflow and its state and interacts with the services layer. Controllers use [routers] to determine the navigation between views. The router maps the URL in a HTTP request onto a template which may itself contain nested templates. Each template is bound to a model and when navigating between views (templates) ember updates the URL provided to the browser (and shown in the navigation bar).

The web application may not directly access the database via ORM. Instead it should integrate solely with the services layer via REST.

8 Mobile application framework

The *Android Application Framework* will be used for the mobile application framework because,

- *Android* devices are the most widely used mobile devices,
- Publication of an *Android* application is less complex and with less strings attached than that of developing and publishing IOS applications.
- Development frameworks are widely available and supported by the open-source community.

9 Reporting framework

For the reporting architecture, *JasperReports* is to be used. It is an efficient, mature and open source reporting framework which is well maintained, widely used. It support the generation of various document formats as well as the generation of images.