A Multi-Purpose Recommender Framework

Halil I. Köklü

Supervised by Nigel Martin

April 2014

A Project Proposal Submitted to
Birkbeck College, University of London
in Partial Fulfillment of the Requirements
for the Degree of MSc Business Technologies

Department of Computer Science & Information Systems
Birkbeck College, University of London

Contents

List of Figures

1 Introduction

In this project I will develop an architecture for the integration of recommender systems.

Essentially, a *recommender system* is software which suggests useful items to a user usually on a given website. A user is typically assisted in their search process on the website to find the right item. Recommender systems are able to personalise so that they try to suggest only items relevant for the given user. In order to do this recommender systems rely on multiple techniques, notably *collaborative filtering* and *content-based filtering*.

Recommender systems are usually tightly coupled to the context and architecture of the website. Bespoke recommender systems may be very dependent on the website's database structure as well as technical constraints such as programming languages. This project aims to develop and evaluate an ecosystem for various recommenders which hides the complexity from external systems and ultimately makes it easy to integrate.

1.1 Proposal Structure

Section 2 introduces the concept of recommender systems. An overview of recommender techniques as well as algorithms are described. Finally, adoption of recommender systems in research and enterprises as well as existing solutions are evaluated.

Section 3 discusses challenges of recommender system as well as requirements for the project. The aims and objectives of the project are explained.

Section 4 presents the architectural design of the solution and defines the technologies to be used. Finally, a project plan is laid out.

2 Background Research: Recommender Systems

This section introduces the fundamental concepts and techniques of recommender systems. Then, it gives two examples of algorithms commonly used in recommender systems. Finally, it discusses the adoption of recommender systems in research and enterprises as well as evaluates some existing solutions.

2.1 Concepts

Ricci et al. (2011) write that traditional recommendations can be observed in various scenarios, such as a peer's recommendation when buying a book or reviews when choosing a movie. The authority of the recommender has an important role in the acceptance of the recommendation. A renowned film critic may appear more credible than a random colleague. When it comes to car parts, a mechanist may be a good candidate to ask. However the authority is not only limited to expertise – in fact we tend to rely more on recommendations which put our personal experiences and preferences into account. A companion for a previous trip can certainly have a better authority than a travel agent.

With the growth of the Internet, the amount of information available on the Web increased rapidly. Especially, major e-commerce Web sites were extending their range of products and services. Although a wider and varied range of items is initially good for the user, users found it more and more difficult to find the appropriate items or make the right choices. Web sites have deployed different type of solutions – such as search engines and more user friendly interfaces – to cope with this problem.

Another approach are recommender systems which basically provide a bespoke collection of items with the intention to highlight relevant items to a user. Depending on the recommendation technique used, various data sources are taken into account like the user's context or previous interactions. This is a continuous learning process. The more the recommender system learns, the more accurate the recommendations will become. The user's behaviour on given recommendations are further a powerful learning source for the system – e.g. if the user tends to accept some recommendations over others – to tweak the recommendations. Most recommender systems concentrate on guiding the user towards novel, unexperienced items (Herlocker et al., 2004).

2.2 Techniques

In the course of development different techniques to compute recommendations emerged. Fundamentally techniques are classified by the information sources they use. The sources of personalised recommendations are typically user-item interactions (*collaborative filtering*), item features (*content-based filtering*), user features (*demographic filtering*) as well as knowledge about the user and item (*knowledge-based filtering*). Non-personalised recommendations also exist in form of ranked lists such as top sellers or related items. However they are typically not part of the research in recommender systems.

Anand and Mobasher (2003) differentiate between *explicit* and *implicit* data collection. Information the user intentionally provides to the system to express a positive preference to items are referred to as *explicit* data collection including rating an item or adding an item to a wish list. *Implicit* data on the other hand is collected by observing the user's behaviour e.g. usage of navigation and search elements or purchase of items.

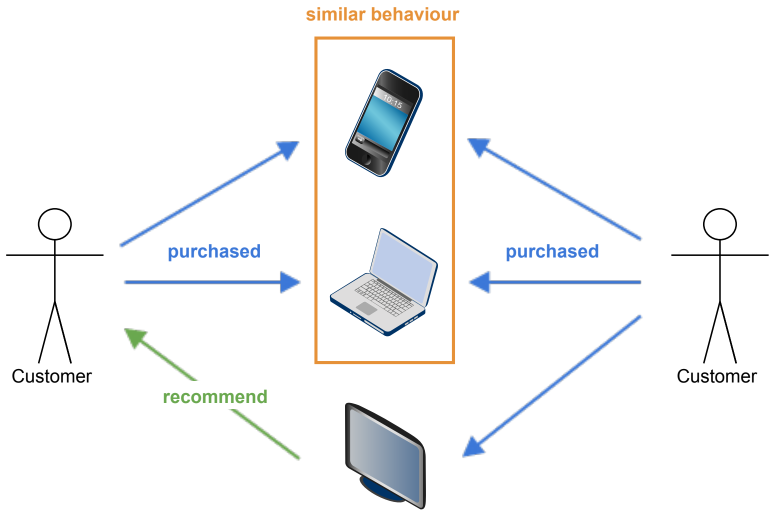


Figure 2.1: Collaborative Filtering

2.2.1 Collaborative Filtering

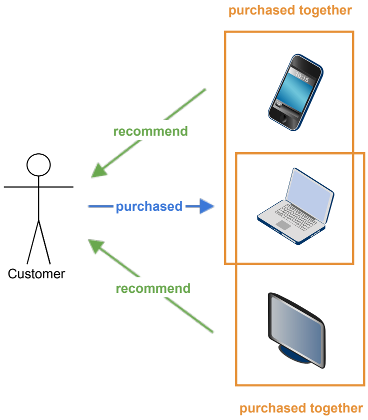
This technique recommends items other users with similar preferences have shown a positive expression e.g. liked or purchased. In order to do this the recommender system needs to observe users' behaviours and interactions. Based on these learnings, it will then look up other users with similar behaviour patterns and build recommendations from their preferences – preferably items which the active user has not experienced yet.

The major advantage of the collaborative filtering approach is that the recommender system does not require any knowledge about the items.

Figure 2.1 illustrates a scenario where the active customer has purchased several items in the past. The recommender system understands that the active customer is similar to another as both have purchased the phone and the laptop. Then, the system computes items the similar customer purchased but the active customer has not. It therefore recommends the TV to the active customer.

Item-to-Item Collaborative Filtering

This approach is a derivative of traditional collaborative filtering methods and has been popularised by Amazon (Linden et al., 2003). The fundamental difference lies in the fact that the item-to-item approach collects collaborative data to put



purchased together	Item #1	Item #2	Item #3
Item #1		1	0
Item #2	1		1
Item #3	0	1	

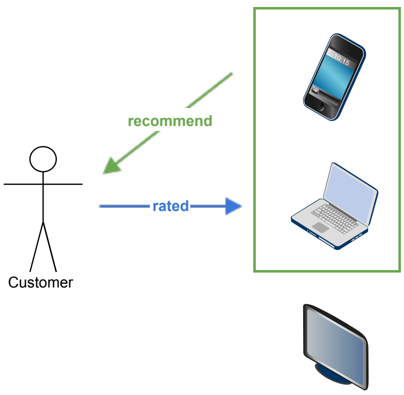
Figure 2.2: Item-to-Item Collaborative Filtering

items into relation with other items rather than users. The recommendation query takes an item as an origin and fetches relative items (see figure 2.9).

Given that the item range can be very wide, item-to-item filtering methods require significant computing time and data storage. However they are usually preprocessed offline thus queries can be processed rather quickly.

Figure 2.2 demonstrates a customer who has purchased an item in the past. The recommender system looks up items which were purchased together with that item and recommends them to the customer.

2.2.2 Content-Based Filtering



Product Data	Category	Brand
Item #1	Mobile	Apple
Item #2	Laptop	Apple
Item #3	Monitor	Samsung

Figure 2.3: Content-Based Filtering

Content-based recommendation methods make use of item features to find similar items. Based on items the user has shown a preference to before – such as rated or purchased – other similar items are looked up based on the item's features. Figure 2.3 illustrates a customer who has rated an item positively. The recommender system compares the rated item with other items, finds another item which has the same brand and therefore recommends that item.

2.2.3 Demographic Filtering

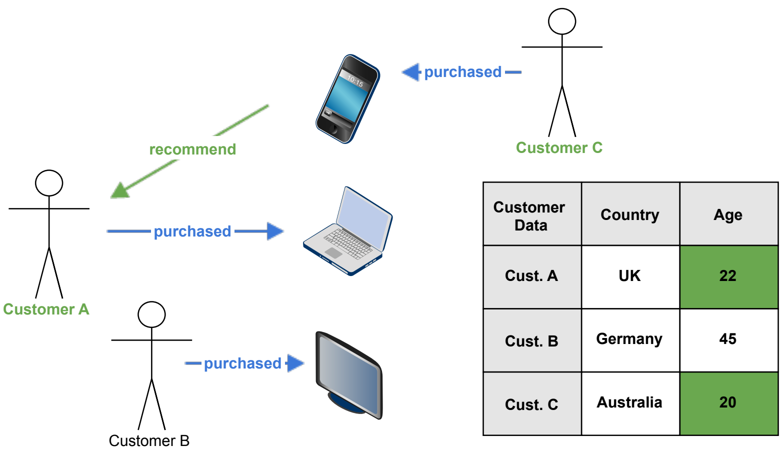


Figure 2.4: Demographic Filtering

The demographic filtering is similar to the content-based method with the significant difference that demographic filtering examines user features rather than item features to find similar users and recommend items those users have shown a preference to in the past. Burke (2007) makes the assumption that recommendations should be different for demographic groups. The demographic profile can consist of age, gender, interests, language, country etc. To give an example, a hotel search engine may want to recommend hotels to a business person and different ones to a young couple.

Figure 2.4 shows how a demographic filtering recommender system would recommend an item to customer A based on a customer base of three customers. The recommender analyses the customer profile features and eventually decides relying on the age feature that customer C is similar to customer A. Hence the system recommends a product customer C purchased in the past.

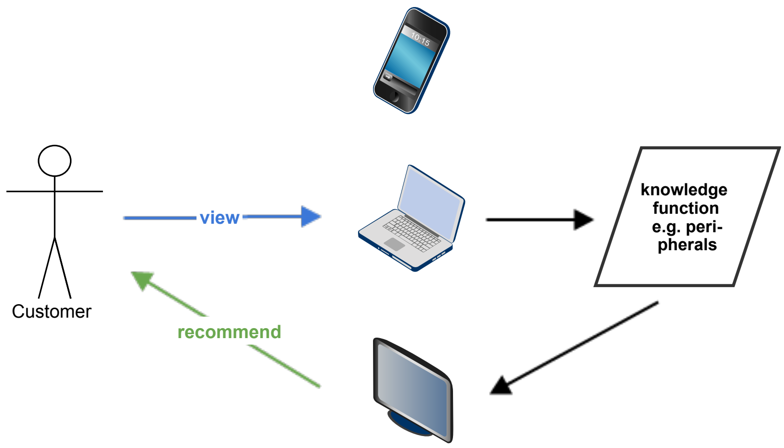


Figure 2.5: Knowledgebased Filtering

2.2.4 Knowledgebased Filtering

Recommendations of knowledge-based systems rely on specific domain knowledge to determine useful items for a user. These systems are usually constraint-based or case-based. Both approaches are similar in their conversational process. In other words, the user specifies the requirements and the system tries to find solutions which are fulfilling them. Whereas constraint-based systems are matching the requirements explicitly (such as price ranges), case-based systems make use of similarity measures (e.g. distance to a point of interest) (Ricci et al., 2011).

An example of a knowledge-based recommender is figure 2.5 which recommends a monitor as a peripheral to a customer who views a laptop. This is based on explicit knowledge that amongst others external monitors are a demand for laptop users.

2.2.5 Hybrids

Hybrid recommender systems make us of two or more individual recommender systems – hereinafter referred to as components – to combine aforementioned techniques. One possible motivation of using hybrid systems may be to overcome weaknesses of one approach by combining it with another. However it is also possible to combine systems implementing the same technique but e.g. using different knowledge sources. There are many possible ways of combining techniques of which Burke (2007) identified seven:

Weighted The scores of recommender components are combined using a linear formula. A score is a numerical rank attached to items.

Switching The system chooses one component over another which is based on some criterion.

Mixed Recommendations from components are summed up and presented together. Implementations usually differ on how the results are merged together.

Feature Combination A contributing component modifies the features of a knowledge source and feeds into the actual recommender component.

Feature Augmentation Similar to *feature combination* this approach's contributing component adds new features rather than modifies them (see figure 2.6).

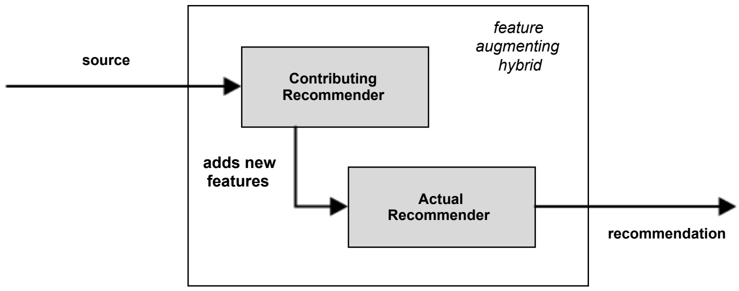


Figure 2.6: Feature Augmenting Hybrid Recommender System

Cascade Components are given strict priorities with the lower prioritised ones breaking ties of the higher ones.

Meta-Level Similar to *feature combination* as well as *feature agumentation*, yet the contributing component completely replaces – instead of appends or modifies – the initial source.

2.3 Algorithms

Recommender systems rely on algorithms. Usually defined to solve a specific problem, it is crucial to use the right algorithm to get the expected results. In the next two sections I will illustrate two algorithms which are very popular in recommender systems.

2.3.1 Pearson Correlation Coefficient

$$Pearson(a,b) = \frac{\sum r_a r_b - \frac{\sum r_a \sum r_b}{N_{ab}}}{\sqrt{(\sum {r_a}^2 - \frac{(\sum r_a)^2}{N_{ab}})(\sum {r_b}^2 - \frac{(\sum r_b)^2}{N_{ab}})}}$$

Figure 2.7: Pearson Correlation Coefficient between ratings by the users a and b where r_a respectively r_b are rating vectors for the mutually rated items N_{ab}.

The *Pearson correlation coefficient* provides a measure to calculate the correlation of two variables. In the recommender systems context it is useful to compare user's behaviour patterns or preferences (Segaran, 2007). The coefficient is defined to be between 1 to -1 where 1 indicates a perfect correlation, 0 no correlation and -1 a perfectly inverse correlation.

2.3.2 Tanimoto Coefficient

$$T = \frac{N_{ab}}{(N_a + N_b - N_{ab})}$$

Figure 2.8: Tanimoto Coefficient where N_a respectively N_b is the count of properties of a respectively b and N_{ab} is the count of intersecting properties.

The *Tanimoto coefficient* tells us the similarity of two sets (Segaran, 2007). It can be used to measure how similar two items or users are based on their features which is useful for *content-based* and *demographic filtering* methods. Below is an example of two items which we want to compare:

```
A = [mobile, apple, iphone, black, 32G]
B = [mobile, apple, ipad, black]
```

Given the equation in figure 2.8 we come to the conclusion:

$$T = \frac{N_{ab}}{(N_a + N_b - N_{ab})} = \frac{3}{5 + 4 - 3} = 0.5$$

2.4 Adoption

Ricci et al. (2011) note that research on recommender systems is relatively new compared to other classical information retrieval methods like databases and search engines. However it is gaining attention due to various reasons. First of all e-commerce companies including Amazon, YouTube and Netflix invest a lot in recommender systems. As trendsetters and pioneers for other institutions they build a demand for further research and development. Dedicated conferences and academic courses as well as its adaption in several academic journals are indicators of recognition of this research area.

E-commerce companies running recommender systems hope among others to increase number of items sold, sell more diverse items, increase customer satisfaction and recommend sequences or bundles (Herlocker et al., 2004).

Figure 2.9: Item-to-Item Recommendations by Amazon

2.5 Evaluation of Existing Research and Solutions

As mentioned in the previous section, recommender systems are considered a relatively new research area in computer science. Although it has gained attention recently, a considerable amount of research has been published on the fundamental concepts and techniques rather than on matters of integration and architecture. Nonetheless I found that Cortizo et al. (2010) and Rack et al. (2007) worked on architectural considerations:

Cortizo et al. (2010) built a *general purpose multi-algorithm* recommender system to compute recommendations from different sources and serving multiple applications. They mention that they could not find any literature on the system's aspects of recommender systems. The advantage of their work was that they evaluated and tested their system on a live environment. Therefore scalability and performance were key metrics from the beginning. Their recommender system is not open source thus unavilable for me.

Rack et al. (2007) have published a series of papers around their work on the *AMAYA* recommender system. Although they have put an emphasis on multi-purposeness, their work is more oriented towards a context-aware recommender system which differentiates between contexts such as *'being home'* and *'being at work'*. Furthermore their recommender requires a user profile as a centre point. In that point their architecture is biased. The last paper about *AMAYA* was published in 2007 and the system is not publicly available.

There are several commercial recommender software available such as *prudsys Realtime Decisioning Engine (RDE)* which I have integrated into a major e-commerce website in the past. In a brief analysis of commercial software I found that they usually cover only basic but common requirements for e-commerce websites. These products are closed source as well.

Hahsler (2011) and Rack et al. (2007) provide a list of open source recommender systems which are freely available. In the majority these systems are more component libraries than complete solutions. They require significant amount of work to integrate. Amongst them is *Apache Mahout* – a machine learning library which also includes collaborative filtering components. *easyrec* on the other hand is a complete solution. However – similar to commercial products – it is opiniated towards e-commerce websites.

3 Problem Definition

This section outlines key problems the project will address. Then, aims and objectives are defined and explained.

3.1 Project Requirements

The project requirements are based on identified challenges of recommender systems. The requirements *interoperability*, *abstraction* and *ease of integration* will become measures on how successful the project is in overcoming those issues.

3.1.1 Multi-Purpose & Interoperability

A *multi-purpose* recommender system is able to cope with different data sources and techniques. Ideally this system is open and extendable for possible forthcoming, yet unknown requirements. It is not to be confused with hybrid recommender systems (see section 2.2.5) which combine different systems into one.

Another requirement is the interoperability between recommender systems. Manouselis and Costopoulou (2007) differentiate between three criteria:

Interoperability of the recommendation queries which allows the same query to be reused. Adomavicius et al. (2005) go further and create a *Recommendation Query Language (RQL)*. The use of an RQL would go beyond the scope of this project. Therefore I will limit this criterion to query parameters rather than the full query and rely on the recommender system to build the query based on these parameters. To give an example, the query parameters for figure 2.3 would be the customer reference and the rated item (in this case the laptop).

Interoperability of the user and the domain models which allows the exchange of models and other data among different recommender systems. In this project I will go a step further and allow direct access to persistence layers to all recommender techniques.

Interoperability of the recommendation results which empowers recommender systems to reuse the results. This is in particular useful for hybrid recommender systems which feed the same query into different systems. Yet there is no use case in this project and therefore will not be covered.

3.1.2 Abstraction

This requirement suggests to separate a software or architecture into smaller components. In this section the process of abstraction is illustrated based on a tightly-coupled, example architecture. With every discussed problem the architecture will become one step closer to an abstracted architecture.

Complexity Applications as well as recommender systems tend to be complex. If not separated, they both add to the overall complexity which makes change management very difficult, time consuming and expensive. Any change requires knowledge, testing and probably modifications on both systems. Figure 3.1 shows a tightly coupled system whose one recommender system is getting replaced. The whole primary application as well as the database are affected.

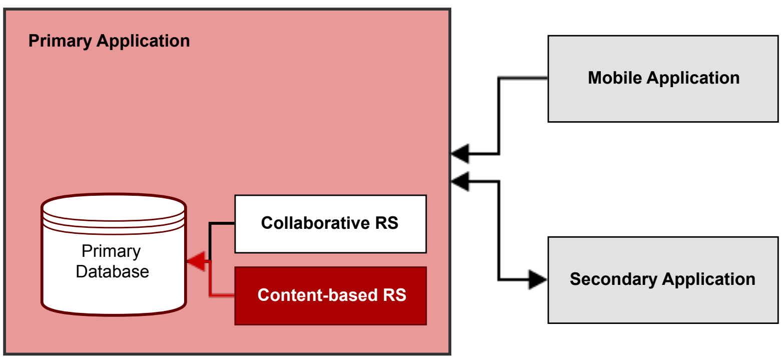


Figure 3.1: Abstraction: Complexity – when the recommender system is replaced or changed, all red components are affected.

Database Abstraction Recommender systems directly using databases of the primary application are problematic as these systems are affected by any change in the database. Figure 3.2 illustrates a semi-decoupled architecture where the recommender system is outside of the primary application, yet still uses its database. This gives a false impression of loose coupling. Teams working on the primary application might not be fully aware that their changes affect the recommender systems.

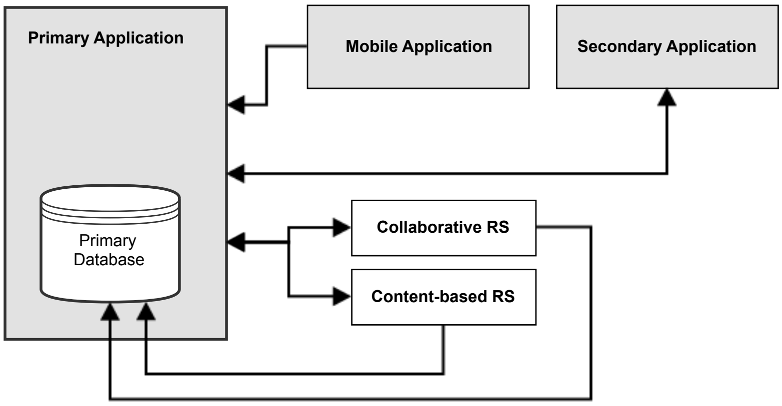


Figure 3.2: Abstraction: Missing Database Abstraction

Reusability Tightly coupled components are difficult to reuse. Figure 3.3 shows an architecture where other applications and even devices such as mobile can reuse the recommender systems.

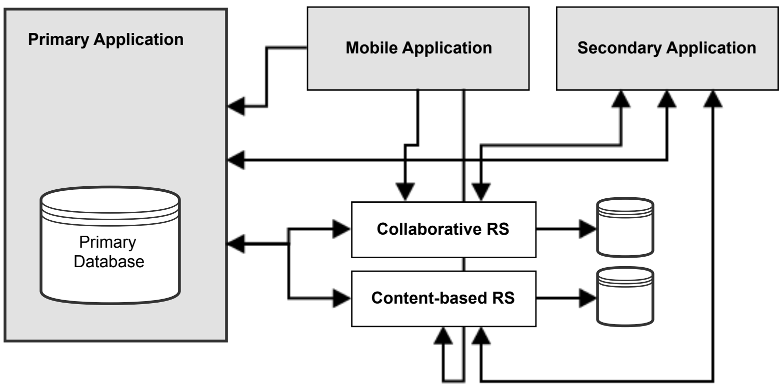


Figure 3.3: Abstraction: Reusable Architecture

Dependancy If a system needs to be replaced or modified, it can have more or less major implications to other systems depending on that system. Figure 3.4 shows a loosely coupled architecture where the recommender technique has been changed to demographic filtering. In contrary to figure 3.1 the primary application is not entirely affected. However the replacement can still affect other systems if the communication between them included content-based filtering specific logic.

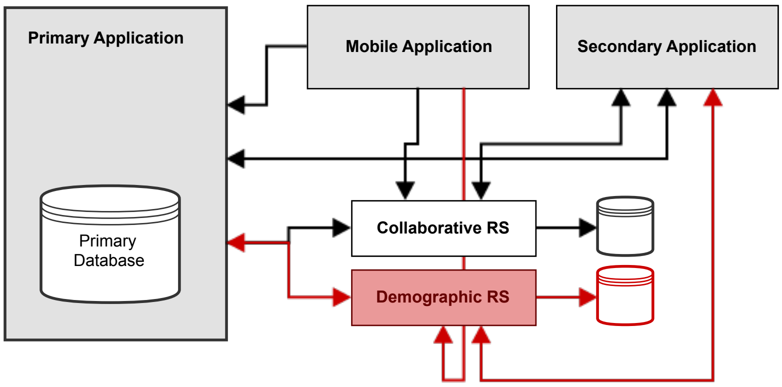


Figure 3.4: Abstraction: Dependancy Issues – when the recommender system is replaced all dependancies (red arrows and elements) might require changes as well.

Encapsulation Information hiding is the fundamental motivation for encapsulation. The more information and implementation is hidden, the looser the coupling becomes. Figure 3.5 illustrates a recommender system framework which hides internal details and communicates with other components in a technique-independent manner. A change of a technique within the framework or an external application should not have any implications.

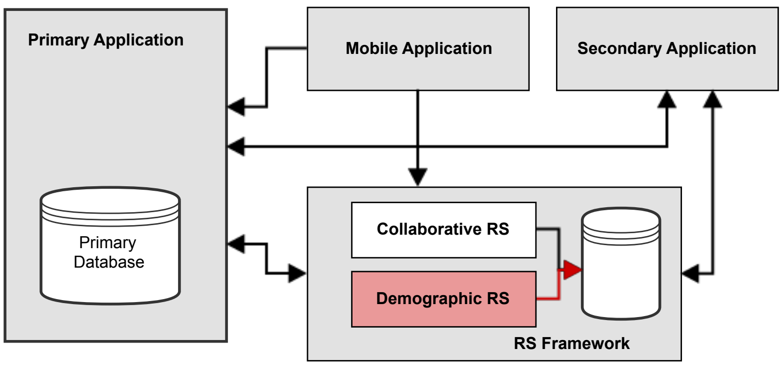


Figure 3.5: Abstraction: Encapsulation – when the recommender system is replaced no other component is affected.

3.1.3 Ease of Integration

The complexity and cost of integration of recommender systems into existing applications is a major constraint for many projects. Ease of integration is therefore a key requirement for this project.

3.2 Aims & Objectives

This project aims to develop and evaluate an ecosystem for various recommenders which hides the complexity from external applications and ultimately makes it easy to integrate.

The primary objective is to design an architecture which fulfills the aforementioned requirements of interoperability, abstraction and ease of integration. The objective is to reduce types of communication between integrating application and recommender systems to an elementary minimum. This hides the complexity of the system.

Another main objective is to build a mechanism which allows the integration of various recommender techniques to coexist and cooperate.

Finally, the practicability and suitability of the design is critically evaluated.

4 Plan for Developing the Solution

In this section an overview of the design and architecture of the project is provided. Then, choices on the technology such as programming languages and databases are explained. Finally, a project plan is laid out.

4.1 Design & Architecture

The architecture of this project is designed to meet the requirements discussed in section 3. First, a bird's eye view is given on where this recommender framework fits into existing systems. Then, the internal architecture of the framework is explained.

4.1.1 Service-Oriented Architecture (SOA)

A *service-oriented architecture (SOA)* is a software design pattern which is most suitable to meet the *abstraction* requirement discussed in section 3.1.2. *SOA* suggests to express features as services. This is true for features which are going to be available to other systems. Internal features are never to be exposed and allowed to be services. A *service consumer* is a system using the service. (Erl, 2008) identifies eight principles of *SOA* of which I will elaborate four:

Standardised service contracts is an expression of the service's purpose, capabilities and requirements – such as mandatory parameters and data types. As long as the requirements are satisfied, the service agrees to fulfill its purpose.

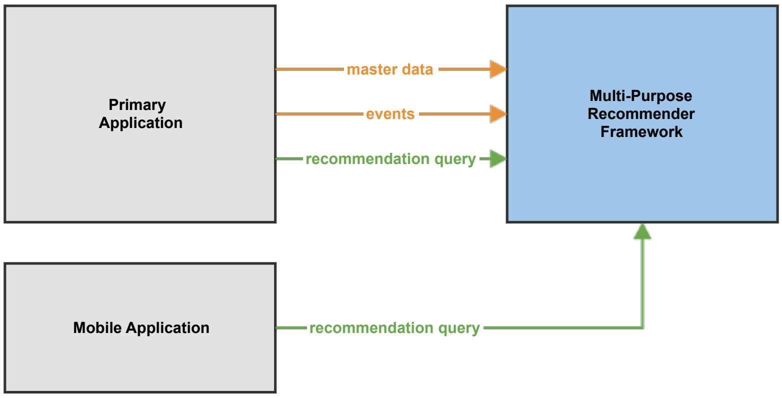


Figure 4.1: Service-Oriented Architecture (SOA). Orange arrows (*master data* and *event*) are *notifications* whereas green arrows (*recommendation query*) are *queries* and expect a result.

Service loose coupling makes sure that services have as few dependancies as possible.

Service abstraction ensure that as much information as possible hidden and none except those described in the service contract are exposed.

Service reusability assure that services are designed to be reused.

In figure 4.1 a possible architecture of the project is shown. The framework is designed to serve more than one application or system component. The services accept requests from any source as long as they authorise with an access key. In the mentioned figure services are divided in two types – *notification* and *query*. A *notification* is a message relevant to the recommender framework and is simply acknowledged as received. A *query* on the other hand expects a result. The framework will define three services:

Master Data enables service consumers to create, update or delete a data node in the framework. An identifier and type are mandatory fields. The service consumer is allowed to send any further features of the node which it thinks is relevant to the framework.

Event accepts notifications about interactions or preferences between two or more nodes such as *'X purchased Y and Z'*. The payload – content of the message – may contain a *weight* which is a numeric value. It is useful for e.g. *'X rated Y with 10'*.

Recommendation Query requests recommendations for a specific *recommenda-*

tion model. The recommendation model identifier is mandatory. A node identifier and type are only mandatory if the model excepts it. This is the only service which expects a result.

The recommendation framework only expects incoming requests (*push strategy*) and has no outgoing communication at all.

4.1.2 Multi-Layered Architecture

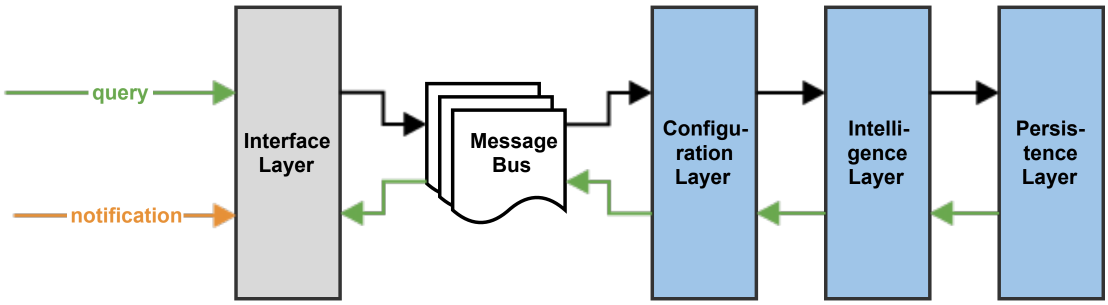


Figure 4.2: Multi-Layered Architecture

A multi-layered architecture is a software design pattern for software programs which suggests separating functionality in responsibility layers. This is popular way of abstracting functionality within a system.

Strictly speaking, the framework is divided into two subsystems: the *application programming interface (API)* and *recommender ecosystem*. These subsystems are connected via message bus. The motivation of this differentiation is of a technical rather than logical nature. The subsystems have different kind of technical requirements and this division allows me to use respectively the most appropriate technology. Another side effect is that a message bus allows me to group and prioritise messages.

Interface Layer

The interface layer is where services expose themselves to service consumers which are those components sending requests to this layer. From a technical point of view this layer is implemented as an *application programming interface (API)* over the *hypertext transfer protocol (HTTP)* which is the foundation of data communication in the world wide web. The *API* adopts the *representational state transfer (REST)* style. An *API* which uses *REST* is called *RESTful*.

REST makes use of *uniform resource locators (URLs)* and *HTTP vocabularies* which is of importance for this project. The *URL* provides a way to locate a resource. The *HTTP vocabulary* defines amongst others *GET*, *POST* and *DELETE*. Combined a *RESTful API* enables the access and modification of resources. E.g. to submit an event a *POST* request to */events* is necessary. To delete a node with the identifier *120* a *DELETE* request to */nodes/120* is sufficient, whereas a *GET* request to */recommendations/topseller* would return all recommendations for a recommendation model called *topseller*. By using low-level *HTTP* vocabulary the *API* requires less documentation and explanation (Fielding, 2000).

This layer acts as *service broker* and *security agent*. Latter is ensured by verifying the presence and correctness of an access key which is a random text. As a *service broker* this layer validates the payload against mandatory fields and data types. Then, it offloads the message and submits it into the message bus for further processing. If the message is of the type *notification* it acknowledges the request. If the message type is a *query*, then it waits for a response from the message bus and returns that.

Configuration Layer

This layer is the heart of this project and targets to satisfy the multi-purpose and interoperability requirement (as defined in 3.1.1). The motivation behind this layer is to set up a recommender system by configuration only. The configuration layer consists of two fundamental ideas: the *recommendation model* and *event*.

A *recommendation model* is an instruction to the recommender on how to compute recommendations. In the model configuration identifier, recommender type and optional parameter are specified.

As mentioned in section 4.1.1, an event is the information about an interaction between two or more nodes such as *'X purchased Y and Z'*. The event subsystem provides a mechanism to allow recommendation models to listen to events. When an event message is received, the subsystem will inform listeners. This way recommendation models receive new data and one event can be used by more than one recommendation model. In a traditional architecture the application would need to update every single recommender system. However the event subsystem makes redundant API calls obsolete as only one message is sufficient. Figure 4.3 shows an example configuration.

```
<config>
    <rule name="items_bought_together" recommender="item_amazon">
        <on event="order_created" do="update" />
        <on event="product_removed" do="remove" />
    </rule>
    <rule name="similar_customers" recommender="collaborative">
        <on event="product_rated" do="add" />
        <on event="order_created" do="add" />
    </rule>
    <rule name="similar_products" node_type="product"
        recommender="content_based" />
</config>
```

Figure 4.3: Recommendation Model Configuration

Recommendation Layer

In the *recommendation layer* holds the actual recommenders which are associated to recommendation models by the configuration layer. The recommender are going to be unaware of the nature of the data. In the contrary it will work based on the parameters fed by the configuration layer. Then, the algorithm will build its algorithm-specific query and use the persistance layer to fetch data and eventually return recommendations.

A distinctive feature of this project will be the recommender plug-in subsystem. All recommenders will be designed as extensions.

Persistance Layer

This layer provides database abstraction components which will then allow any recommender system to access any of the supported storage systems.

4.2 Technological Choices

In this section I will discuss my choices in technology I want to use in this project.

4.2.1 Application Programming Interface (API)

The *API* is the interface layer in this project and defines almost none business logic. In that sense it is important to use a lightweight, thin solution. As discussed in 4.1.2 I will use a *RESTful* approach for the *API*.

The *API* will be built on the *node.js* platform which makes use of *Google Chrome*'s fast *V8 JavaScript engine*. The *API* will further use *express* – a web application framework for *node.js*. *express* is ideal for *RESTful APIs* as it uses the *HTTP* vocabulary as well. Figure 4.4 illustrates a sample implementation of '*GET* request to */recommendations/topseller*' mentioned in 4.1.2. As visible in the figure the footprint of the implementation is very thin. As the *API* has a specific format in *express*, it is able to automatically generate *API* documentation.

```
app.get('/recommendations/:id', function(req, res){
    // do work
});
```

Figure 4.4: Sample API for recommendations in *express*

The message format will be in *JavaScript Object Notation (JSON)* - a thinner alternative to *extensible markup language (XML)*.

4.2.2 Message Bus

The concept of the message bus was introduced in section 4.1.2. The main requirement is that it implements the *advanced message queuing protocol (AMQP)* – an open standard which defines a minimum set of features in message buses. A popular, highly reliable message bus software which supports *AMQP* is *RabbitMQ*. It also provides a *graphical user interface (GUI)* giving information about the current message flow. I have worked with *RabbitMQ* in the past.

4.2.3 Recommender Ecosystem

This subsystem contains configuration, intelligence and persistence layers. It needs to be performant due to the implementation of recommenders which can require a lot of computing resources. In that sense dynamic scripting languages such as *hypertext preprocessor (PHP)* or *Python* are not my first choice. In fact a statically-typed, compiled language with concurrency capabilities is needed.

My preferred candidate for this is *Go* – a language developed by *Google* as an alternative to overcome limitations of the programming language *C++*. The result is a language which is intendedly not adopting all patterns found in many programming languages such as overloading and pointers. The language aims to be simple, safe and free of misinterpretations by the compiler. It supports multithreading, CPU paralleling as well as asynchrony. Finally, it has a modern package management system which allows retrieving packages from the internet.

Database Software

The recommender ecosystem requires a powerful yet flexible database software. As pointed out in section 4.1.1 a master data can have an arbitrary number of additional fields. To fulfill the *ease of integration* requirement a schema management solution is not desired. Schema-less databases – also known as *NoSQL* – might be of interest. *NoSQL* databases are usually non-relational as well. However recommendations are very intensive in terms of relations (e.g. *'X rated Y'*).

A potential solution are graph databases which understand *nodes* and *edges* – relations to nodes. A *property* can be attached to *nodes* as well as *edges*. Latter is useful for weighted relations such as *'X rated Y with 10'*. Figure 4.5 shows a sample graph. Graph databases have further advantages especially in querying distant *nodes* via other *nodes*. With regard to figure 4.5, an example query could be to fetch all groups people, who Alice knows, are members of.

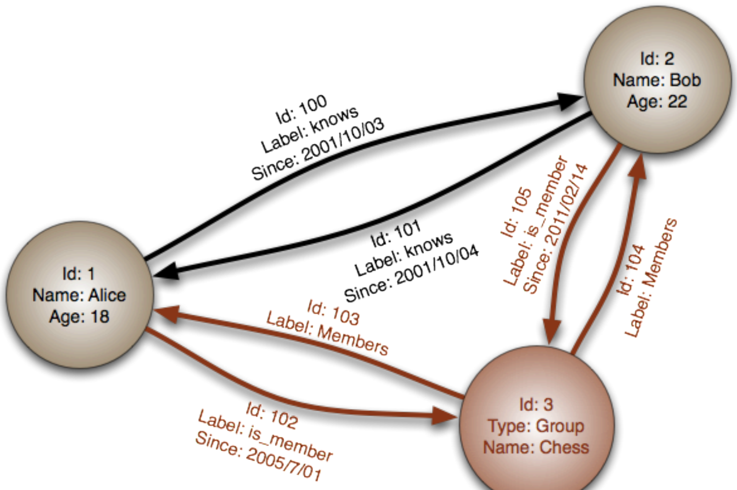


Figure 4.5: Simple Graph in a Graph Database. Source: Creative Commons.

4.2.4 Others

A *version control management (VCS)* namely *git* will be used throughout the project. *Git* is a distributed *VCS* which is amongst others faster and more flexible than other *VCS* such as *Subversion*. The *git* repository – and therefore the source code – will be hosted on *BitBucket* to have a backup of the project anytime.

For server health and performance monitoring I will use the *software as a service (SAAS)* solution *NewRelic*. It also allows to analyse specific low performing or faulty requests. This will be very useful during testing and evaluation of the solution.

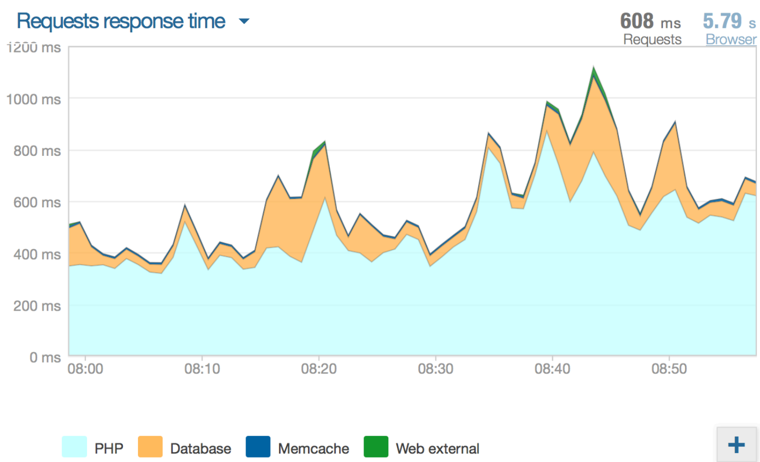


Figure 4.6: Monitoring with NewRelic

Finally, a virtualisation solution called *VirtualBox* will be used to set up all technical and vendor requirements within a virtual machine. This prevents conflicts with dependencies on the workstation especially if versions differ. When submitting the project the virtual machine will be packaged to provide a seamless demo setup for examination.

4.3 Demo

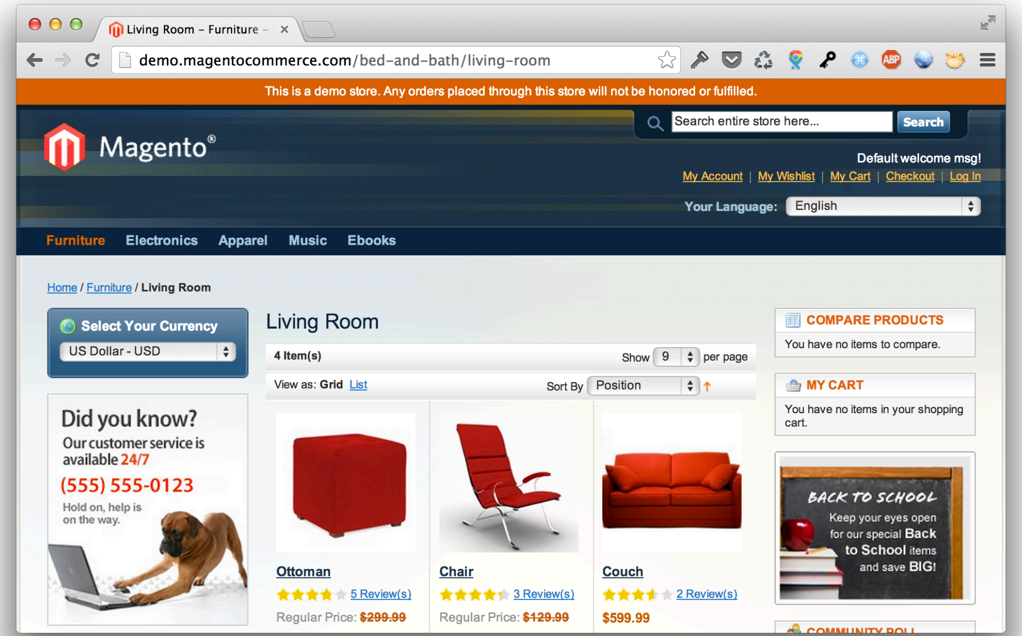


Figure 4.7: Open-Source Ecommerce Web Application *Magento*

The proposed architecture primarily deals with internals. Although it can be tested by simulating *API* calls, it is not very visible and tangible. Therefore the solution will be demonstrated on a platform which reflects a real-life use case. As seen in the background research most applications using recommender systems are e-commerce websites.

It is hence not surprising that the demo will be as well an e-commerce website which will be based on the open source e-commerce web application *Magento* – strictly speaking the free community edition. Having used *Magento* professionally I am comfortable extending it for my testing purposes. The product database will be loaded with ca. hundred thousand individual products which allows us to evaluate basic scalability and performance testing.

4.4 Project Plan

This section provides a work schedule to complete the project as well as a fallback plan in case of unexpected or time-related circumstances.

4.4.1 Schedule

The project will be split into two broad phases: *implementation* and *evaluation*. The implementation phase is from 5th May to 10th August 2014, then the evaluation phase is until 15th September 2014. As the project has significant implementation work to be done, this strict division makes sure that enough time is left for the evaluation phase.

The schedule in figure 4.8 is divided into bi-weekly time frames in which one or more tasks are to be delivered.

4.4.2 Fallback Plan

In this section alternative paths are defined in case of delays due to underestimated work or other external circumstances.

The software architecture is the critical path of the project. Little is gained when layers or distinctive features are missing. Therefore I will focus on delivering them. In case of delays it is possible to decrease the number of different techniques and recommendation uses cases. In the worst case scenario, a the requirements of recommendations can be kept low so that I can use simpler recommender systems.

Go and *Neo4j* are yet unfamiliar to me. In case of problems, they can be replaced by simpler or more familiar choices. *Go* itself could be replaced by *PHP* or *Python*. Instead of *Neo4j* the key-value store *Redis* may be a simple alternative.

	Time Frame	Tasks
Implementation	5th - 18th May	Set up virtual machine as well as Magento. Implement interface layer. Implement publishing to message bus.
	19th May - 1st June	Implement master data service.
	2nd - 15th June	Exam Break
	16th - 29th June	Implement event service.
	30th June - 13th July	Implement plug-in system for recommender techniques.
	14th - 27th July	Implement recommendation model service.
	28th July - 10th August	Implement several recommender techniques.
Evaluation	11th - 24th August	Evaluation of architecture and recommendation quality. Structure report.
	25th August - 7th September	Write chapters of the report related to implementation and architecture.
	8th - 15th September	Write remaining chapters of report and finalize. Write documentation.

Figure 4.8: Project Schedule

5 References

Adomavicius, G., Tuzhilin, A., and Zheng, R. (2005). Rql: A query language for recommender systems.

Advanced Message Queuing Protocol (AMQP) (2014). http://www.amqp.org/. Accessed April 07, 2014.

Amazon.com (2014). http://www.amazon.com/. Accessed April 07, 2014.

Anand, S. S. and Mobasher, B. (2003). Intelligent techniques for web personalization. *Proceedings of the 2003 international conference on Intelligent Techniques for Web Personalization*, 7(4).

Apache Mahout (2009-2014). https://mahout.apache.org/. Accessed April 07, 2014.

BitBucket (2014). https://bitbucket.org/. Accessed April 07, 2014.

Burke, R. (2007). Hybrid web recommender systems. In *The Adaptive Web*, pages 377–408. Springer Berlin Heidelberg.

Cortizo, J. C., Carrero, F. M., and Monsalve, B. (2010). An architecture for a general purpose multi-algorithm recommender system. In *Workshop on the Practical Use of Recommender Systems, Algorithms and Technologies (PRSAT 2010)*, page 51. Citeseer.

easyrec (2009-2014). http://www.easyrec.org/. Accessed April 07, 2014.

Erl, T. (2008). *Soa: principles of service design*, volume 1. Prentice Hall Upper Saddle River.

express (2014). http://expressjs.com. Accessed April 07, 2014.

Extensible Markup Language (XML) (2014). http://www.w3.org/XML/. Accessed April 07, 2014.

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California.

Git (2014). http://git-scm.com/. Accessed April 07, 2014.

Go (2014). http://golang.org/. Accessed April 07, 2014.

Google Chrome (2014). http://www.google.com/chrome/. Accessed April 07, 2014.

Hahsler, M. (2011). recommenderlab: A framework for developing and testing recommendation algorithms. *Nov*.

Herlocker, J. L., Konstan, J. A., Terveen, L. G., and Riedl, J. T. (2004). Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems*, 22(1):5–53.

HTTP Vocabulary in RDF 1.0 (2014). http://www.w3.org/TR/HTTP-in-RDF10/. Accessed April 07, 2014.

JavaScript Object Notation (JSON) (2014). http://www.json.org/. Accessed April 07, 2014.

Linden, G., Smith, B., and York, J. (2003). Amazon.com recommendations: Item-to-item collaborative filtering. *Internet Computing, IEEE*, 7(1):76–80.

Magento (2014). http://magento.com/products/overview. Accessed April 07, 2014.

Manouselis, N. and Costopoulou, C. (2007). Analysis and classification of multi-criteria recommender systems. *World Wide Web*, 10(4):415–441.

McCarthy, J. (2007). What is artificial intelligence? http://www-formal.stanford.edu/jmc/whatisai/whatisai.html. Accessed April 07, 2014.

Neo4j (2014). http://www.neo4j.org/. Accessed April 07, 2014.

NewRelic (2014). http://newrelic.com/. Accessed April 07, 2014.

node.js (2014). http://nodejs.org/. Accessed April 07, 2014.

Prudsys (2014). prudsys realtime decisioning engine. http://www.prudsys.de/en/solutions/rde-recommendations.html. Accessed April 07, 2014.

RabbitMQ (2014). https://www.rabbitmq.com/. Accessed April 07, 2014.

Rack, C., Arbanowski, S., and Steglich, S. (2007). A generic multipurpose recommender system for contextual recommendations. In *Autonomous Decentralized Systems, 2007. ISADS'07. Eighth International Symposium on*, pages 445–450. IEEE.

Redis (2014). http://redis.io/. Accessed April 07, 2014.

Ricci, F., Rokach, L., and Shapira, B. (2011). *Introduction to Recommender Systems Handbook*. Springer US.

Segaran, T. (2007). *Programming Collective Intelligence: Building Smart Web 2.0 Applications*. O'Reilly Media, Inc.

V8 JavaScript Engine (2014). https://code.google.com/p/v8/. Accessed April 07, 2014.

VirtualBox (2014). https://www.virtualbox.org/. Accessed April 07, 2014.