University of Groningen
Faculty of Mathematics and Natural Sciences
Department of Computing Science
Software Patterns

Software Architecture Document

Pattern-based Design

Group 2
Buck, Werner
Heijkens, Jan-Alexander
Ioakeimidis, Spyros
Manteuffel, Christian
Katsantonis, Fotis
Hanjalic, Avdo

0.4
Groningen, July 16, 2012

Authors

Name	Acronym	E-Mail
Ioakeimidis, Spyros	si	spirosikmd@yahoo.com
Manteuffel, Christian	cm	cm@notagain.de
Katsantonis, Fotis	fk	fotiskatsantonis@gmail.com
Hanjalic, Avdo	ah	a.hanjalic@student.rug.nl
Buck, Werner	bk	wernerbuck@buck@gmail.com
Heijkens,Jan-Alexander	jh	jaheijkens@gmail.com

Revision History

Version	Date	Description
0.1r	11/23/11	Set up structure based on document template Introduction on Business model and work on System Context
0.2r	11/30/11	Changed outline HL-1, HL-4, functional requirements functional and non-functional requirements architectural decisions - patterns functional and non-functional requirements, architectural decisions - patterns
0.3r	12/07/11	Initial Model Elaborated Model Evaluation Update Evolution First FRMs Changed Pattern Decision Template
0.4	12/14/11	Changed outline Improved Initial Model Adopted Entire System Description Adoption System Over and Evolution Adopted Elaborated Model Improved FRMs and Evaluation

Legend: r = reviewed

Contents

List of Figures

List of Tables

1 Introduction

This document describes the software architecture of the eTicket system. This document is created during the course Software Patterns at the University of Groningen.

The eTicket system is a ticket reselling platform that allows event organizers to offer tickets and event visitors to purchase tickets. The system covers the complete process of selecting a ticket, seating, subsequent payment and ticket verification at the entrance of the event itself.

Since the organization of an event is not an easy task, handling the ticket pre-selling is something that organizers do not want. Such an advance sale of tickets requires to address several problems, which includes a secure payment process, reliable and verifiable tickets as well as handling peak request phases for top-events. Solving these problems is a costly and complex task for a single event organizer. The eTicket system should address the mentioned problems, so that event organizers can rely on the provided service and focus on organizing great events instead of handling the selling process on their own. We at eTicket want to take away any aspects of ticketing from the event organizer.
The advantages for event visitors are that they get a standardized process of purchasing tickets with fixed general terms and conditions and a high emphasis on usability. In general, the eTicket platform is a mediator between Event Organizers and Event Visitors, as illustrated in Figure 1.

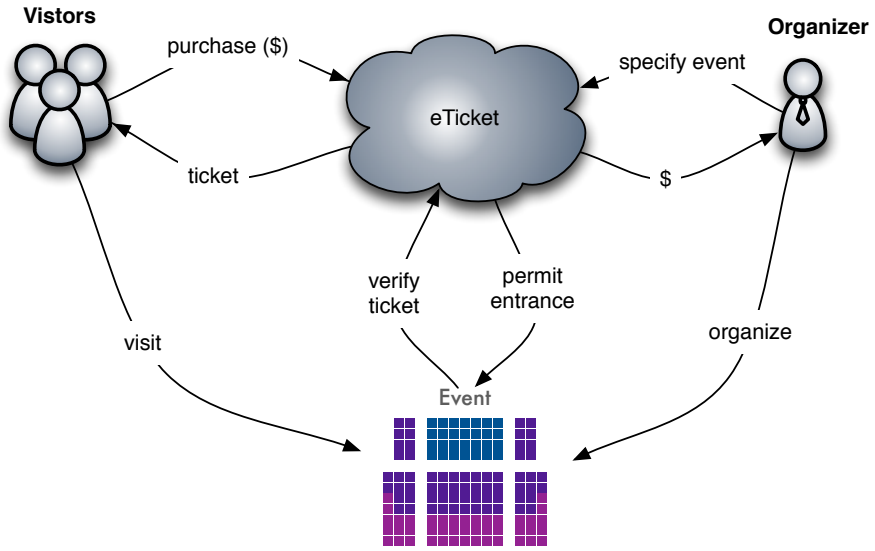

Figure 1: eTicket mediates between Visitors and organizers

The architecture document is structured as follows. At first an overview of the system is provided (cf. Section 2) including a high-level description of the functionalities and purpose, the stakeholders of the system as well as their concerns and the resulting key drivers.
After that the requirements are stated, separated into functional and non-functional requirements (cf. Section 3). In Section 4, the architectural decisions are presented, in this case the choice for particular patterns that support the system qualities and functions. Subsequently, the orchestration of the patterns that forms the actual software architecture is presented in Section 6. The document finishes with an evaluation of the elaborated architecture (cf. Section 7) and an outlook of the evolution of the system (cf. Section 8).

2 System Overview

eTicket is an online ticket reselling platform for arbitrary events. It aims to provide a consistent platform for event visitors to purchase tickets and to relieve organizers from the responsibility to organize the online ticket pre-selling on their own. Figure 2 presents an overview of the system.

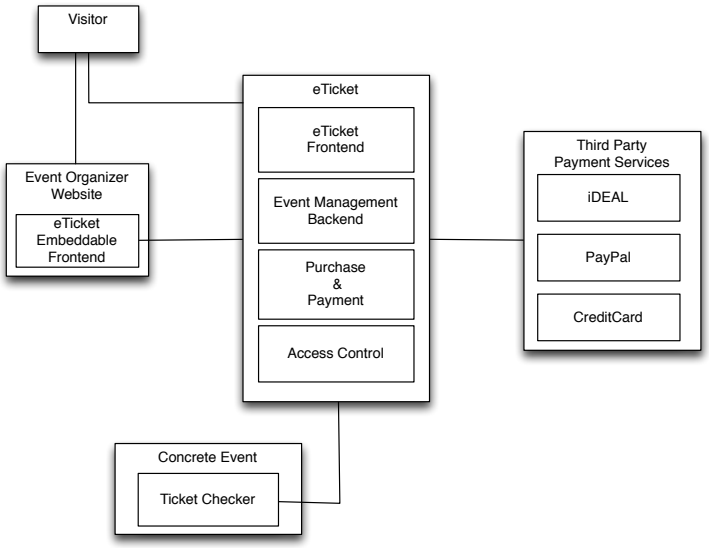

Figure 2: System Overview

The eTicket platform is profitable by taking a share on the original ticket price. For example, the event organizer aims to sell a ticket at a price of 20€. When the visitor buys the ticket for 20€, the eTicket system takes a share of 10% (2€) before transferring the remaining 18€to the account of the event organizer.

The system provides a web-interface for Event Organizers (SH-02) to create an event, where they have to provide all necessary event information. They have a special user account with which they can gain access to a secure site.

Organizers can specify different price categories for tickets. Furthermore, the appearance of the ticket backend can be customized, because it will be embedded into the Organizers own website. Organizers have the possibility to provide a seat planning for an event, but this is optional. eTicket provides a number of generic seat plans that the event organizer will be able to use. If the generic seat plans do not match the needs of a specific event, the event organizer can provide a seat plan of his own. The creation of this seat plan will be standardized and eTicket will supply a guide with specific steps to follow to create a seat plan, to make life easier for the event organizer. To make the system better aware of its environment and all the venues, custom seat plans provided by the event organizers will be stored in the systems data storage for future use. Next to this functionality from the event organizers perspective it should be possible for customers to purchase tickets. This service can be accessed in two ways, as illustrated in Figure 3:

- Via an embedded eTickets component on the website of the event organizer.
- Via the eTicket Portal, which lists all events registered with eTicket.

The Ticket buying functionality is embedded into the Event Organizer website via IFrame. Hence, no data will be exposed to the Event Organizer during the purchasing process.

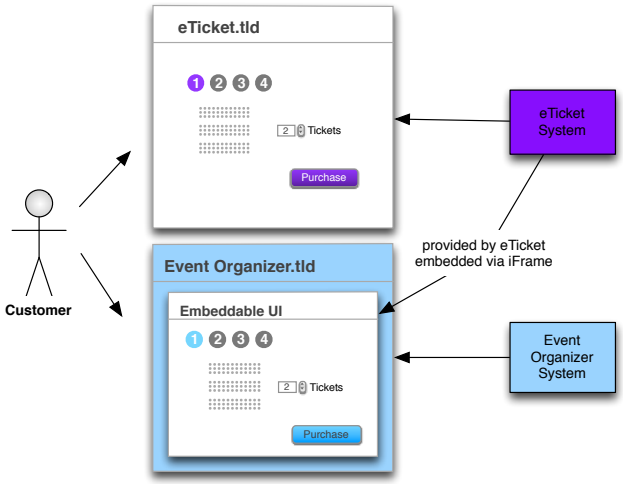

Figure 3: Event Organizer can embed the functionality to buy Tickets in their own website.

Independent on whether the customer uses the embedded or portal service, he can authenticate himself with his central eTicket user account, that allows to quickly purchase tickets without the necessity to provide all information again. Though it is necessary to register the first time. The benefits of the central user account become visible when more and organizers start using the eTicket system.

The system can be accessed on a personal computer[1] or mobile device[2]. Tickets will be distributed to the customers via email. The customer can print the ticket or use a special mobile ticket. Both ticket variants use a QR code. By using our own ticket format, the system shall provide a possibility to verify tickets at the entrance of a concrete event. The tickets are actually scanned by devices that use software produced by eTicket. The tickets are scanned by these devices, which match a ticket with the database on site at the event.
The customer can pay a ticket either via PayPal, CreditCard and because the system will be launched in the Netherlands also iDEAL is supported. Supporting these three external payment services will increase customers confidence in the system, because they are widely used and well-known. Purchasing a ticket, involves transferring the money safely from the customer to the event organizer. When registering an event a bank account needs to be provided, this bank account will be checked according to the IBAN number of that account. During a transaction the system automatically subtracts the stipulated share of the original ticket price. For security [KD2] of the customer the total money will be payed to the event organizer after the event took place. In this way malicious event organizers can create an event but will not receive the money. The money can then easily be transferred back to the customers. The method and time of transferring this money can be defined by the event organizer and eTicket Inc.

[1] A normal computer has a screen resolution of at least 1024x768 and a physical keyboard
[2] e.g Devices that run the operating system iOS or Android.

2.1 Stakeholders

In this section the stakeholders and their concerns are discussed. For each stakeholder the name, formal identifier and priority are given, followed by an elaboration on the stakeholder's concerns.

Customers (SH-01, High)

The customers are using the system to buy tickets for events. They want the process of purchasing tickets to be easy, while it has to be secure at the same time, as money is of concern. Furthermore, they are concerned that the system works reliable and that the purchased ticket are correct.
Summarized, the concerns of the customers are:

Performance: customers expect the system to respond quickly to user input.

Reliability: customers expect the system to work as intended.

Security: transactions and account details have to be processed/stored in a secure way.

Usability: to prevent customers refraining from a purchase due to not being able to work with the system, the system has to be easy to use.

Event Organizer (SH-02, High)

The Event Organizer plans new events. In order to sell tickets via the eTicket system he/she creates new events by providing all the necessary event information.
The event organizers want an affordable solution that is highly available, enabling customers to purchase tickets at any time. They presume that the system produces valid tickets. Furthermore, the service should be easy to embed in the merchant's web-site.
Summarized, the concerns of the organizers are:

Affordability: the system has to be affordable in order to arouse the interest of organizers.

Integrability: it has to be easy for organizers to integrate the ticket-selling mechanism in their web-site.

Performance: customers should get the tickets within a certain time bound, otherwise chances are great they will refrain from purchasing the tickets.

Reliability: organizers should receive the correct amount of money for all sold tickets.

Security: transactions and account details have to be processed/stored in a secure way.

Payment Handler (SH-03, Low)

The payment handler is the company that provides the service for online payments like Credit Cards institute, PayPal and iDEAL. They want the eTicket system to connect with their services efficiently so they can handle as many transactions as possible. Furthermore, they have interest that eTicket works reliable and is secure.
Summarized, the concerns of the payment handlers are:

Performance: to enable many transactions, the system has to be fast.

Security: transactions and account details have to be processed/stored in a secure way.

Developer (SH-04, Medium)

This group develops the eTicket product. It contains Software Architects, Engineers as well as Testers. The developers want a product that is fun to develop and can be easily maintained and tested. They want to deliver a stable product within the time boundary.
Summarized, the concerns of the developers are:

Maintainability: the system has to be designed with eye for maintenance, in order to reduce effort required to update and/or extend the product in the future.

Testability: a well testable product should decrease total development time and effort.

Product Owner (SH-05, High)
This is the the company that is the owner of eTicket and will provide it as a service to the Event Organizers (SH-02). They want a profitable system, which should work as promised because failures will cost money and harm the name of the company. Additionally, they want a product that is very reliable and available 24/7. Altogether, they want a profitable product which satisfies customers and event organizers.
Summarized, the concerns of the product owner are:

Maintainability: for updating and/or extending the product in the future it has to be maintainable.

Performance: customers should get the tickets within a certain time bound, otherwise chances are great they will refrain from purchasing the tickets.

Profitability: to maximize profitability it is important to make a cost-effective system that can handle a lot of workload.

Reliability: the system administration should always conform the given output to all system users to prevent them from losing their confidence in the system.

Security: transactions and account details have to be processed/stored in a secure way.

Event Personnel (SH-06, Low)
The event personnel are the people checking and/or selling tickets at the event. They want to be able to check the validity of tickets easily, while having confidence in the (verification) output given by the ticket checking system.
Summarized, the concerns of the event personnel are:

Performance: checking tickets should be fast as it is a repeated operation.

Reliability: verification of tickets has to be 100% water-proof as the ticket checkers to rely on it.

Usability: checking tickets should be easy as it is a repeated operation.

Negative Stakeholders

Hackers (NSH-01, High) This group tries to jeopardize the privacy of customers (SH-01) by stealing personal information. They are also expected to be interested in booking money from eTicket's bank account to their own accounts. Hackers have a negative interest in **Security**. Therefore eTicket system's security must be tight to ensure that customer information and eTicket's bank credit are secure.

In Table 2 the stakeholder concern matrix is illustrated. The matrix lists the previous mentioned concerns and sums them up based on the priority of the stakeholders. Each stakeholder is free to assign points wherever he finds it necessary. For stakeholder priorities the following weights are defined: High = 100, Medium = 50 and Low = 10.

Table 2: Matrix of stakeholder concerns

(By priority) **Stakeholder** \ **Concerns**	Usability	Security	Reliability	Affordability	Integrability	Performance	Maintainability	Testability	Profitability
Customer	100	100	100			100			
Event Organizer		100	100	100	100	100			
Product Owner		100	100			100	100		100
Hacker		100							
Developer							50	50	
Payment Handler		0	10			10			
Event Personal	10		10						
Total	110	**400**	**320**	100	100	**310**	150	50	100

2.2 Key Drivers

The following key drivers are deduced from Table 2.

Security: As the system deals with large amounts of money, in order to prevent loss of money to malicious parties, security is of great importance. This should be visible throughout the whole system design.

Reliability: The system has to deal with large amounts of money, therefore it is of primary concern that the transactions are administered and processed as intended. If it is not able to process some transaction, (the failing of) the transaction should be consistently processed throughout the whole system.

Performance: The amount of customers and organizers is expected to grow rapidly, which will result in a higher load on the system. However, the system's performance must be high under any circumstances to prevent users refraining from using eTicket. Therefore performance is a key-driver for the system design.

3 Requirements

3.1 High Level Requirements

Re/Selling Tickets		
HL-1	**Must**	An option to embed a ticket selling component shall be provided to event organizers in order to integrate ticket selling into their websites. Customers shall use a login system prior to booking tickets. Existing or newly created customer accounts can be used to book tickets for future events. The distribution of the tickets is done through email (e.g. to display on a mobile phone or print out).

Creating Events		
HL-2	**Must**	The Event Organizer shall be able to create and edit events. The system shall provide an interface for event organizers, in order to create events. The system shall provide a possibility to specify seats, pricing models and seat plans. The system shall provide a number of generic seat plans. Submition of custom seat plans will also be possible.

Handling Payment		
HL-3	**Must**	The system should be able to handle the payment transactions between customers and event organizers. The customer should have the possibility to pay for a ticket through different payment handlers e.g. iDEAL, PayPal or MasterCard and Visa. After payment from the customer, a margin is subtracted from the received amount and the remainings are added to the organizer's credit. The system has to ensure that the payment is secure and reliable even during peak load periods.

Entrance Control		
HL-4	**Must**	Printed and mobile tickets shall be verified by scanning the ticket QR code and verifying it with the system's administration. The ticket checking system shall be able to update validation checks on eTicket's databases, related to events details. The system shall ensure reliability of ticket validations.

3.2 Use Cases

In this section we will briefly go through the most basic use cases for our system. The use cases that have are related to the main eTicket service are shown first. Then we show the use cases that take place at the event.

3.2.1 Use Cases related to eTicket

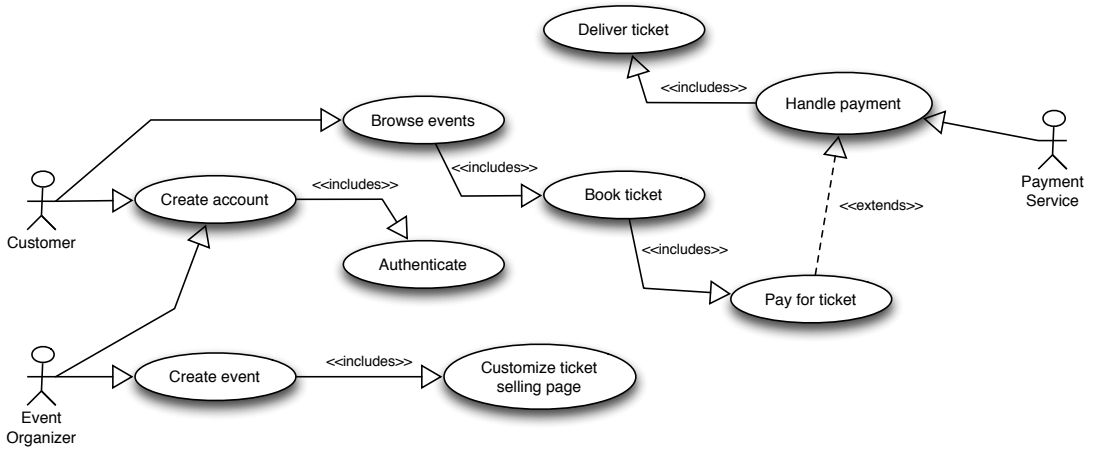

Figure 4: High level eTicket Use Cases

The use cases shown on figure 3 are the basic use cases for our main system. We briefly describe them according to each actor. We start with the event organizer. The most basic use case is that of an event creation. In order to be able to create an event he must first have an account though. Once he is authenticated he can create an event. To do so he must enter the details of the event and he can optionally add more details so more information about the event are displayed. After creating the event he can customize the ticket selling page that will be embeded to his website so it matches the style of his website.

The customer can create an account, which is required to book a ticket. He can browse the available events from our portal without having to register and authenticate. If he wants to book a ticket he needs to be authenticated first. The process of booking a ticket includes selecting the event for which he wants a ticket and paying for the ticket. The payment of the ticket is handled by third party payment services like paypal or iDeal for example. After the payment is complete the ticket is delivered in an electronic form via e-mail.

The payment service actor represents the third parties that we cooperate with to handle payments. Their role is limited since they are only responsible for handling the actual payments.

3.2.2 Use Cases related to Event

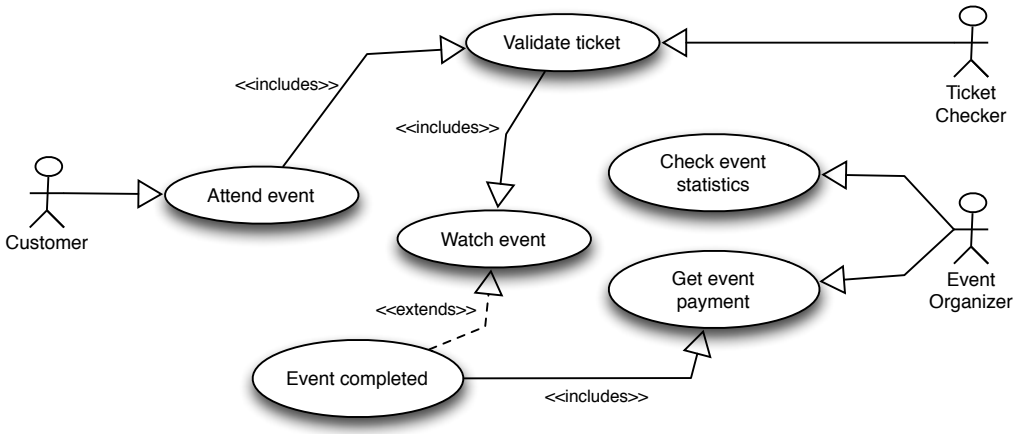

Figure 5: High level Event Use Cases

The use cases shown in figure 4 are the basic use cases that take place at the event. We start with the customer. The customer goes to the event for which he/she has bought a ticket. In order to enter the event his/her ticket needs to get validated at the entrance. After this is done succesfully he/ she can attend the event.

The ticket checker is the one responsible for using the ticket checking devices to validate the tickets of the customers at the entrance of the event.

Finally the event organizer can check live statistics for the event, for example attendance rates. These statistics are generated real time by our system. Because the tickets are validated at the local server that exists at the event we can provide the event organizer with these statistics. After the event has been completed the event organizer receives his payment for the event.

3.3 Requirements

3.3.1 Functional

Ticket Sales

F-1.1	**Must**	The system shall provide user authentication and registration using email and password both for event organizers and customers. The customer and event organizer logins are separated from each other.
F-1.2	**Must**	Customers shall be able to book one or more tickets for an event through ticket selling components.
F-1.3	**Must**	The system shall always send a digital copy of the ticket via E-Mail.
F-1.4	**Must**	The system shall provide an option for selecting a seat during the ticket buying process. When selecting a seat, that seat is locked out for buying for other customers at that moment.

F-1.5	**Must**	The system shall upon registration by a customer ask for details like mobile number, post address etc to reduce the time needed to book a ticket in the future. This information is stored and is used as default information for subsequent ticket bookings. The user has the option to modify these input options.
F-1.6	**Must**	The system shall make note of the tickets purchased by the customer in the system. This information will be available at the user profile.

Managing Events

F-2.1	**Must**	The System should provide a possibility to create and edit events via web-interface.
F-2.2	**Must**	The system shall provide the possibility for event organizers to embed the eTicket portal (a ticket selling component and event overview) for a particular event into their own website.
F-2.3	**Must**	The event organizer must provide at least the minimum amount of information for an event. An example of what the minimum required information shall be is: city, country, address, zip code, phone number, price of ticket, number of available tickets, refunding/cancelation policy, event category, time and date of the event, last time possible to book tickets.
F-2.4	**Must**	The organizer can enter custom information for an event. He will be able to adapt the service according to his needs. Examples of such information can be: seating plan, number of tickets per customer or ticket price variation according to seating.
F-2.5	**Must**	For an event to be created a verified bank account must be associated with the event organizer.
F-2.6	**Must**	The system shall check the validity of a bank account provided by the event organizer to prevent fraud.
F-2.7	**Must**	The system shall provide a set of generic maps of existing event locations for seat planning.
F-2.8	**Must**	The organizer has the option during the edit or creation process of an event to send in a seating plan with subsequent pricing details.
F-2.9	**Must**	The system shall provide the possibility to event managers to change the layout and colors of the embeddable eTicket portal (a ticket selling component and event overview)
F-2.10	**Must**	The system shall not allow the booking of a ticket from a customer after the last possible time specified by the event organizer.
F-2.11	**Must**	The system shall save a number of statistics in the database like for example the rate of tickets being sold, or the number of tickets validated for future statistical use.
F-2.12	Optional	The system shall provide the option to provide default options for creating events (an example). This is optional, only to aid the ease of input for the event organizer.

Handling Payment

F-3.1	**Must**	The system shall support the following payment methods for the customer: PayPal, iDeal, MasterCard and Visa.
F-3.2	**Must**	The system shall be able to abort the payment process at any time.
F-3.3	**Must**	The system shall provide high security for payment transactions with encryption mechanisms and TLS[3] protocol.

Entrance Control

F-4.1	**Must**	The system shall be able to communicate through internet, wifi or lan with ticket validation systems and devices.
F-4.2	**Must**	The system shall be able to validate ticket QR codes in real time. In case of an error in the ticket validation process the system shall inform the validation system.

3.3.2 Non-Functional

Reliability

RELIAB-1	**Must**	The system shall be available 99.9%. $\alpha = \frac{\text{mean time to failure}}{\text{mean time to failure+mean time to repair}} = \frac{3\ years}{3\ years+1\ day}$
RELIAB-2	**Must**	The system shall be able to ensure a reliable (always consistent) ticket validation process.
RELIAB-3	**Must**	The system shall be able to validate ticket QR codes after reading them, in less then 3 seconds.
RELIAB-4	**Must**	In case of an error throughout the process of any kind of transaction, the system shall cancel this particular transaction and inform the user (customer/event organizer) of an error.
RELIAB-5	**Must**	The system shall be able to ensure a reliable (always consistent) payment process.

Security

SECUR-1	**Must**	The system shall be secure, no unauthorized access to system functionality should be permitted.
SECUR-2	**Must**	The system shall be able to protect sensitive data (user data).

Usability

USAB-1	**Must**	The system shall be easy to use for customers, meaning customers should be able to find what they are looking for in minimal number of clicks.

Performance

PERFOR-1	**Must**	Any kind of web request shall be responsive, in the way that feedback is received within five seconds.
PERFOR-2	**Must**	The system shall be able to handle a large amount of customers, wanting to book a ticket, with a maximum of 6000 requests per second during peak times of ticket booking, for one or multiple events.
PERFOR-3	**Must**	The system shall be able to handle a peak load during peak times of ticket validation, for one or multiple events.

Scalability

SCALAB-1	**Must**	The system shall be able to handle network overloading during peak times of ticket booking, for one or multiple events.

Adaptibility

ADAPT-1	Optional	The system should be able to adapt to more third-party payment services in the future.

4 Analysis

4.1 Architectural Decisions

In this section the PDAP[4] method is used for the partitioning of the system. Now that we have established the identification of the architectural drivers, which is mainly extracted from stakeholders concerns and requirements, this section explains the decisions, rationales and alternatives involving software patterns.

4.1.1 Layers

The project will definitely be a web-application. This pattern will help us to decompose it, and provide extra security.

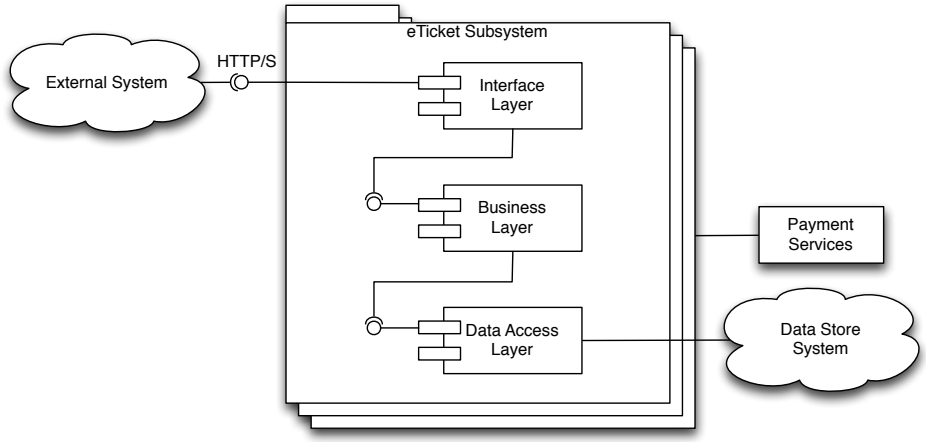

Figure 6: Layers used for system decomposition

Source

Pattern-oriented Software Architecture: A System of Patterns [**?**]

Issue

The system consists of high level components performing tasks like users interaction and authentication as well as low level components which perform tasks like payment and database transactions handling. Although the system is distinguished of a strong relation between those components, decoupling them is crucial. The desired normal flow of data through the system is that requests are moving from high level components to low level components and answers to these requests follow the opposite direction.

Assumptions/Constraints

A constraint is that eTicket is a large system that requires decomposition.

Positions

1. Layers
2. Relaxed Layers

[4]Pattern-Driven Architectural Partitioning

Decision

The system is structured into layers. Each layer provides an interface for the layer on top of it and uses the interface from the layer below. Layers are responsible for processing the http requests, user authentication, payment and database transaction processing as well as handling input from and sending results to event organizers and customers. All components inside each layer will work at the same level of abstraction.

Argument

1. By using layers for structuring the system all tasks are decoupled and grouped by similar responsibility hence increasing maintainability. Layers provide their own interfaces which has a positive effect on security which is the most important key driver of the system. These separate interfaces are capable of enhancing security as different security mechanisms can be used in each interface.
2. In a relaxed layered system a layer can use services from all the layers below of it. A layers may also be partially opaque, meaning that some of its services are only visible to the next higher layer. while others are visible to all higher layers. Although with this variant of layers pattern there is a gain in performance, there is also a significant loss in security. This is the main reason that it is not being used.

Implications

The usage of layers impedes performance. Although generally the requests of the system do not transfer substantial amount of data, in most cases they have to go through all layers in order to be processed and follow the opposite direction to give results. Response times cannot be fast and hence performance is hindered.

Related Requirements/Decisions

F-1.3, SECUR-1, SECUR-2

4.1.2 Model-View-Controller

In the system with the Layers pattern, there is a top layer called Presentation. In this layer we introduce the components needed to implement the Model-View-Controller pattern.

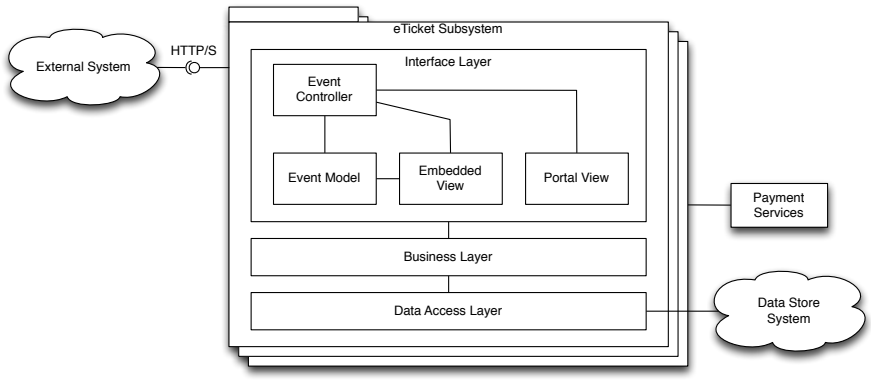

Figure 7: Model-View-Controller Pattern handles different Views of the same Model

Source

Pattern-oriented Software Architecture: A System of Patterns [**?**]
Architectural Patterns Revisited – A Pattern Language [**?**]

Issue

The system has a dynamic web-based user-interface that should be decoupled from the application logic. This user-interface must be able to be adopted depending on the context in which it is used, e.g. on the Webpage of eTicket or embedded into the Event Organizer page. Also, the Event Organizers want an adjusted interface that neatly integrates with the colors and layout of their own webpage.

Assumptions/Constraints

The pattern must be usable in the domain of web-development.

The system contains multiple views. Which view is applied depends on where the particular web-viewer is coming from.

Positions

1. Presentation-Abstraction-Control
2. Model-View-Controller

Decision

Model View Controller will be used to decouple the multiple views of the system from the application logic. It will be used in the interface layer of eTicket subsystems that contain multitple views.

Argument

1. The Presentation-Abstraction-Control pattern structures the user-interface into hierarchical agents. This decomposition is not well-suited for the eTicket System. Furthermore, the increased complexity and lower efficiency impact the overall system's performance negatively.
2. The MVC pattern allows to use multiple views for the same model. As illustrated in Figure 7, the EventModel can be used in both views. Furthermore, the user-interface is decoupled from the system's function by separating view and controller.

Implications

By using MVC for our presentation we have given it the responsibility of generating the view. This means we can add and remove views without worrying about the model, and it improves readability of the code, making changes to the views easier.

Related Requirements

F.1-2, F.1-5, F-1.6, F-2.1, F-2.2, F-2.3, F-2.4, F-2.5, F-2.8
USAB-1, PERFOR-1

4.1.3 Shared Repository

We know that users and tickets should be linked, we need a database. And the database is accessed by different clients so it has to be shared, and consistent among clients. The Shared Repository as will now follow would fit as one of the bottom layers (or even the end).

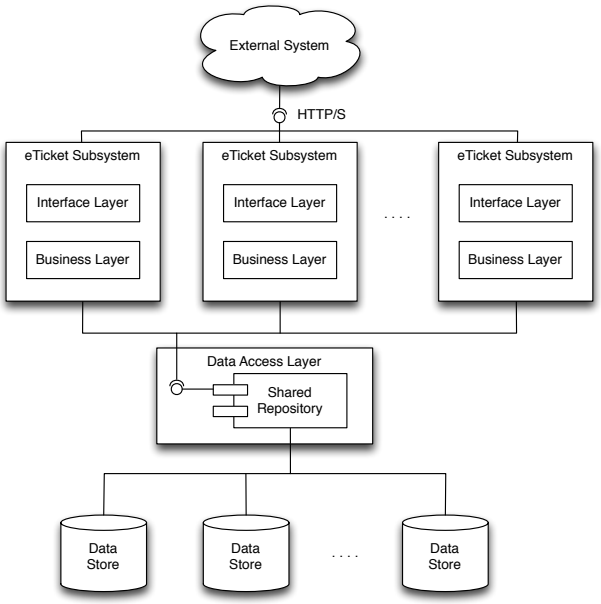

Figure 8: Shared Repository

Source

Architectural Patterns Revisited – A Pattern Language [**?**]

Issue

The four key components namely Event Management, Ticket Sales, Ticket verification and regular content serving all depend on the same data. The pattern needed must be able to handle data contention and be scalable enough to meet the clients requirements. Furthermore, it must ensure data consistency while being shared with multiple components of the system.

Assumptions/Constraints

A constraint is that the repository uses a shared language that is usable for all our components, for example SQL. It is assumed is that the shared repository also handles authentication for access to it.

Position

1. Shared Repository
2. Active Repository
3. Blackboard

Decision

Shared repository will be used as shared data store for all clients to be able to access. Shared repository will be applied in the data access layer.

Argument

1. The shared repository fits, as multiple components must access the same data. And the long-term persistence of our data requires a centralized data management.
2. Active Repository is the variant of Shared repository that notifies on changes. However we do not require this active polling of information in our system, nor is it present in our concerns.
3. Blackboard handles non-deterministic problems. In this system all problems are deterministic, therefore there is no use for such a pattern.

Implications

The Security keydriver is more enforced, as the shared repository is a single point of entry, although it also has the downside of being a single point of failure.
The Reliability keydriver is enforced, as data persistence is insured as all data is shared from one point. However if the server would go down, no data would be available.
The performance keydriver is impaired by this pattern because it only has a single point of entry, making this a possible bottleneck for our system. These forces need to be further resolved to solidify our key drivers. This is resolved with use of the Fail-over Pattern

Related Requirements

HL-1,HL-2,HL-3,HL-4
F-1.5, F-1.6, F-2.1, F-2.2, F-2.3, F-2.4, F-2.7, F-2.8, F-2.11, F-2.12

4.1.4 Service Data Replication

Our requirements clearly state that we need live ticket verification. It is however not feasible to directly contact a distant database from all around the world for performance reasons. For this reason we clearly need a high-level-pattern that describes that we replicate portions of the database.

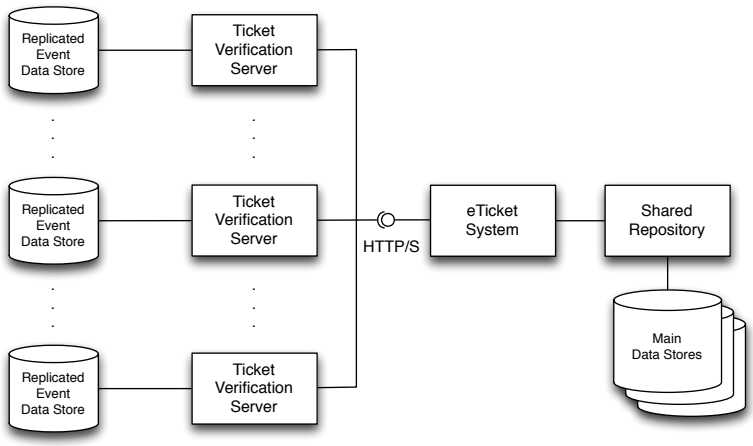

Figure 9: Service Data Replication creates replicas of data store for each event

Source

SOA Design Patterns [**?**]

Issue

Ticket verification processes for each event need to access the main data store in order to verify tickets QR codes. Internet connection can be proved as a bottleneck due to possible failures. Also the response times of the system upon ticket verification requests need to be as fast as possible. However with the use of Layers (Section 4.1.1) performance is impeded which means that response times cannot be fast. Ticket verification requests need to pass through all layers, reach the data access layer and then follow again the opposite direction.

Assumptions/Constraints

Presence of an active internet connection int he area where events take place is a constraint.

Positions

1. Service Data Replication

Decision

Each event will be provided with its own ticket verification server as well as its own replicated data store related to its data. Hence ticket verifications can be realized locally as ticket verification devices can verify tickets QR codes with the replicated data store. Ticket verification servers on each event will be connected to eTicket service in order to be able to replicate events related data from the main data store.

Argument

1. Autonomy is increased and strain on the main data store is reduced. Reliability is increased because there is no dependency on internet availability as far as ticket verification processes are concerned. Also performance is significantly increased, as fast response times upon ticket validation processes can be achieved.

Implications

Applying this pattern will result in additional cost and demand as far as infrastructure is concerned. Also an excess of replication data stores may be difficult to manage. Furthermore security may become an issue as new infrastructure is proposed which also operates separately from the main system infrastructure. Probably additional security measures need to be taken.

Related Requirements/Decisions

F-4.2, RELIAB-2, RELIAB-3, PERFOR-3

4.1.5 Broker

The devices used for ticket verification should be able to use any form of communication to access the local server which uses the Service Data Replication pattern above.
Therefore we define the Broker that is also a Client Server variant.

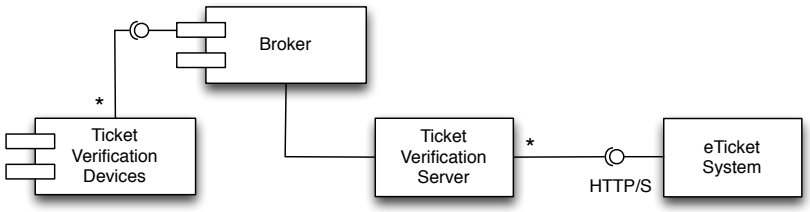

Figure 10: Broker

Source
Architectural Patterns Revisited – A Pattern Language [**?**]

Issue
The ticket checking hardware should be kept as simple as possible to reduce cost, as there could be a lot of ticket checking machines. However if the ticket-checking machine would have to store the tickets that it has checked, and still needs to check. And make all those comparisons themselves, the device would be very expensive, as performance would be an issue.

Assumptions/Constraints
We assume the QR code is read correctly and that communication is possible between the client (the ticket checking device) and the server (the local machine at the event site that contains all the tickets)

Position

1. Client Server
2. Remote Procedure Call
3. Broker
4. Message Queueing

Decision
Broker will be used between ticket verification servers and ticket verification devices for the realization of the communication between them.

Argument

1. The client server is a very general pattern that actually encapsulates broker, it is in fact so general that we do not use it.
2. Remote Procedure Call is very similar to Broker, however broker is more simple and only has the responsibility to deliver an object and await status. There is no need to make it seem like the remote object is executed on locally.
3. The broker pattern helps provide a way to execute code on a remote object, while ensuring safe transit by means of marshaling. It hides and mediates all communication problems.

Effectively, broker allows the ticket-checker to remain simple, and only function as a communications/scanning device. While the server at the other end is capable of processing all tickets and making sure this happens consistently.

4. Message Queuing is not necessary as the server can handle more than one request at a time. Also the message size would be trivial and the complexity of the ticket checking software would increase.

Implications

The ticket-checking software should not have to worry about different protocols or ways of communication, and there is more focus on passing along the information and keeping the programming simple.

Related Requirements

HL-4, F-4.1, F-4.2, RELIAB-3

4.1.6 Brokered Authentication

Security for our main site and embedded component is a key driver. For this reason we introduce the following pattern that describes a way to authenticate securely.

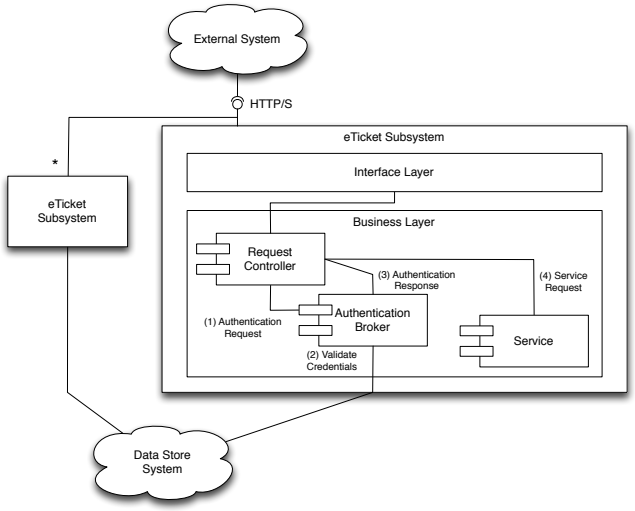

Figure 11: Brokered authentication

Source

SOA Design Patterns [**?**]

Issue

Access to secure areas of the system needs to be restricted. The user will have to authenticate himself with our system in order to be able to access those areas.

Assumptions/Constraints

We assume that the user wants to access a protected service on our system, thus he has to authenticate.

Positions

1. Brokered authentication
2. Direct authentication

Decision

We decided to use the brokered authentication pattern to address this issue. The user will have to authenticate with the authentication broker before he can access restricted areas of the system. After successfully authenticating, the user will be assigned an authentication token which can be used to access the services provided by our system. The tokens will bear the "signature" of the authentication broker so the services can verify that they can trust the particular token.

Argument

1. This pattern enhances security of our system which is the main key driver. By handling security with the security token the user only needs to authenticate once which increases the usability of our system. Furthermore the security token issued by the authentication broker can be used for further authentication checks.

2. This pattern also increases the security of our system. The disadvantage of this compared to brokered authentication is that it does not handle security centrally. Thus if the user wants to access more than one restricted services on our system, he will have to authenticate each time he tries to access a service.

Implications

The brokered authentication is a single point of failure. We can introduce backup or redundant authentication brokers to surpass this problem. However that will also increase the complexity of our system.

Related Requirements

F-1.1, F-1.6, F-2.1, F-2.8

4.1.7 Trusted Subsystem

Specific sections of the website, or uses like payment and booking of a ticket need to be extra secure. For this reason we introduce the following Trusted Subsystem pattern, which uses the above defined Brokered Authentication to increase security.

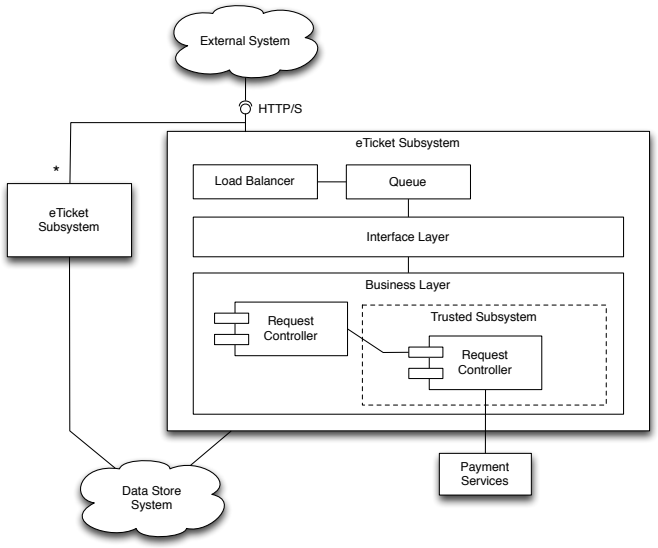

Figure 12: Trusted subsystem

Source

SOA Design Patterns[**?**]

Issue

There is a risk that a customer could circumvent our login service and compromise the integrity of the system.

Assumptions/Constraints

We assume that the user is authenticated via the authentication broker and has a valid authentication token.

Positions

1. Trusted subsystem

Decision

We chose to use the trusted subsystem pattern to enforce security for the concerned areas. It will authenticate the user against its own internal credentials. The trusted subsystem service has a set of credentials that are associated with the specific user/organizer account. When the user tries to access the services that are under the supervision of the trusted subsystem, the trusted subsystem service will verify the identity of the user, using its own credentials. This process will be transparent to the end user.

Argument

1. This pattern enhances security which is our main key driver. It works in a transparent to the user way which does not interfere with the usability of our system.

Implications

Performance is slightly reduced due to the additional overhead. Also if the attacker manages to get to the protected area, he can exploit resources of our system.

Related Requirements

F-3.2, F-3.3, RELIAB-5, SECUR-1, SECUR-2, ADAP-1

4.1.8 Cache Proxy

As can be seen from the requirements and system overview section, the eTicket service will have a lot of requests.
By introducing the following Cache Proxy pattern, and placing the components needed at key locations in our system, the eTicket service becomes faster. As we have a lot of static content like event information etc, that all have an impact on our database.

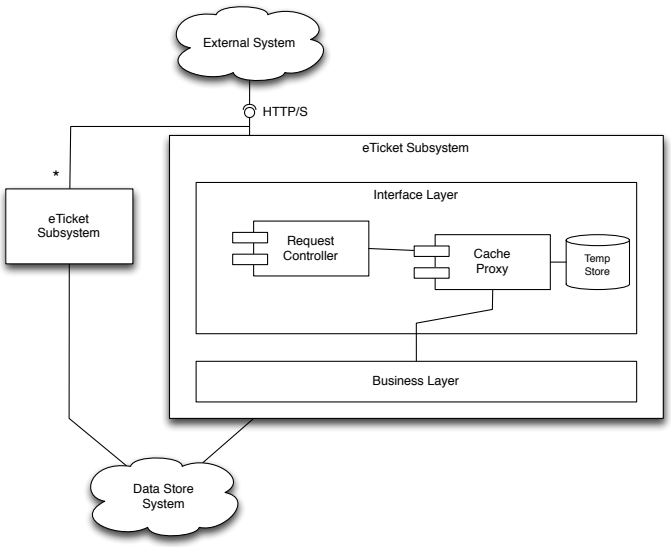

Figure 13: Cache proxy

Source

Pattern-oriented Software Architecture: A System of Patterns [**?**]

Issue

When the site is under load, the database, as being the most intricate component will suffer from every single request being made for data.
These requests can be menial to very important, some could just request event information. And other requests are for buying tickets which are very crucial.
In the event of a large amount of load the users that would want to buy tickets would suffer, as repository data requests can not be prioritized.

Assumptions/Constraints

We expect that most of the user requests will be read requests and not requests to write to the database. We also assume that most of the users will be requesting a set of pages that is frequently accessed. Finally we expect that the content of our pages will not be changing very often since events are not created on a constant rate, but with a rather large interval in between.

Positions

1. Cache proxy

Decision

We will use the cache proxy in front of the portal so it can intercept incoming requests. If the proxy has the requested data already cached it will send the data. Otherwise it will forward the request to the database, store the results on the cache and send the results to the initial request. When the memory of the proxy reaches the maximum limit, a least frequently used strategy will be used. We choose this strategy because we expect most of our users to browse the same pages.

Argument

1. The cache proxy will increase the performance of the system since as we mentioned most of the requests are expected to be read requests. The cache proxy will be handling those requests instead of delegating them to the database.

Implications

The cache proxy is a single point of failure. But the failure of the cache proxy is not mission critical since the requests can still go to the database. Also some data cached on the proxy might be outdated, but that is not the usual case since as we already mentioned, the data that we display are not expected to change very often.

Related Requirements

RELIAB-1, SCALAB-1, PERFOR-1, PERFOR-2, PERFOR-3

4.1.9 Load-Balanced Cluster

The cache defined above is not enough to cover the performance issues we might have when under load. For this reason we introduce the following Load-Balanced Cluster pattern.
By launching copies of systems, we attempt to scale horizontally with the load.

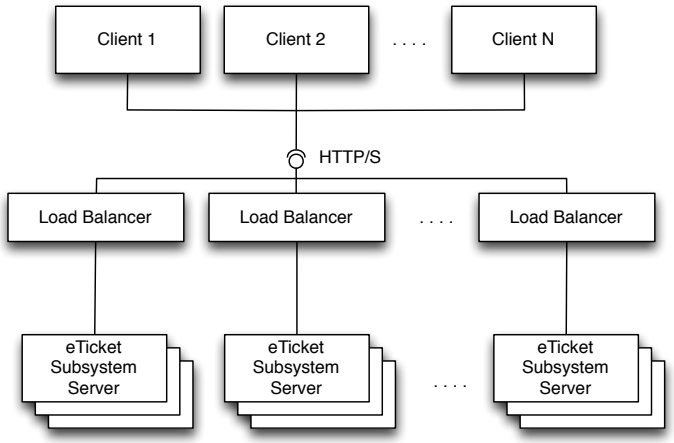

Figure 14: Load Balance pattern

Source

Enterprise Solution Patterns Using Microsoft .NET [**?**]

Issue

It is expected that a large amount of users will be using the system, exhausting the system's

resources, which could render (a part of) the system overloaded. Besides a risk exists that a server crashes or dies due to failing of hardware. To prevent such a scenario it is necessary that the system can scale over multiple servers in order to not only increase the total amount of available resources, but also tolerate a machine to crash due to hardware failure. Furthermore it must be easy to add servers in the future, as the average and peak loads are expected to grow in time.

Assumptions/Constraints

The systems that are to be load-balanced should be capable of working besides each other.

Positions

1. Load Balanced Cluster
2. Server Clustering
3. Failover Cluster

Decision

The system will be designed using the Load-Balanced Cluster pattern. Hardware load-balancers will be used for all components for which this pattern is applicable (see implications). Load-balancing enables the system to maintain acceptable performance levels while enhancing availability. When the system starts showing symptoms of being overloaded, e.g. long response times, resources (servers) can be added easily. Hence the performance and reliability can be maintained in the future and during peak times. Furthermore the system can be realized cheaper by using many slower machines instead of one high-performance server.

Argument

1. The Load-Balanced Cluster specializes in providing a way of horizontal scaling to handle more load, and more efficiently provide a reliable service.

2. The Load-Balanced Cluster is a variant of Server Clustering, specialized - as the name says it - for balancing the load. Server Clustering is applied for distribution of many services, while the goal in this case is to enhance the performance and availability (reliability) of one specific service.

3. The Failover Cluster provides a solution for hardware failing, but does not utilize all servers at the same time, which is necessary to cope with peak loads and also accomplished by the Load-Balanced Cluster.

Implications

Only stateless services can be load-balanced while being protected against hardware failures, unless the state sharing is implemented, as the service state is only contained by the machine it is running on. However, the latter issue is not resolved by this pattern and will have to be implemented separately.

Related Requirements/Decisions

RELIAB-1, PERFOR-1, PERFOR-2, PERFOR-3, SCALAB-1

4.1.10 Failover Cluster

A variant of the previously introduced Load-Balanced Cluster is the Failover Cluster. These two patterns can be used as a combination on a cluster. And this pattern particularly focusses on the Reliability keydriver.

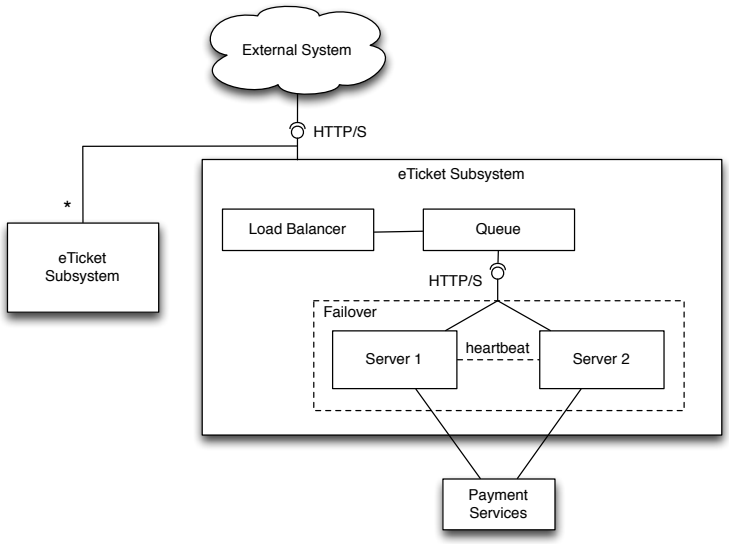

Figure 15: Failover Pattern

Source

Enterprise Solution Patterns Using Microsoft .NET [**?**]

Issue

When performing payment, an user needs the guarantee that his payment is accepted when it is completed. As the payment has to be completed before the system will accept the transaction as successful, failing of a machine at this point can lead to a critical inconsistency: the payment could be completed, when the machine fails, rendering the transaction unaccepted.

Assumptions/Constraints

Assumed is that delays in network traffic are too short to cause inconsistencies in state replication.

Positions

1. Fail-over Cluster
2. Server Clustering
3. Load-Balanced Cluster

Decision

The relevant subsystem(s), e.g. payment handling, will be designed using the Failover Cluster pattern. Multiple (i.e. more than one) servers will be configured as one virtual host to perform the required task. Transaction states are replicated on all servers, by means of *hot standby* state synchronization: updates to the internal state of the active server are immediately copied to the standby server. Hence the (hardware) failure of a machine

will lead to an inconsistent state (e.g. unsuccessful transaction, while the payment was accepted).

Argument

1. The Failover Cluster provides the reliability we need, as opposed to the other patterns described

2. The Failover Cluster is a variant of Server Clustering, specialized - as the name says it - for server failover. The Server Cluster pattern provides a solution for hardware failing, however it does not implicitly handle state synchronization, which is necessary to guarantee state consistency.

3. The Load-Balanced Cluster has the benefit of distributing load over all available machines. Despite its failover capabilities, it is not applicable for stateful services without adding additional logic on all machines for state synchronization. The latter is mandatory to guarantee for transaction based services like payment processing.

Implications

Services protected against hardware failures by the Failover Cluster will not scale easily: hardware load balancing is not applicable without duplicating every machine that is to be load-balanced.

Related Requirements/Decisions

RELIAB-1, RELIAB-5, PERFOR-1

4.1.11 Indirection Layer

The Indirection layer is combined with the Layers pattern to for indirection to payment services.

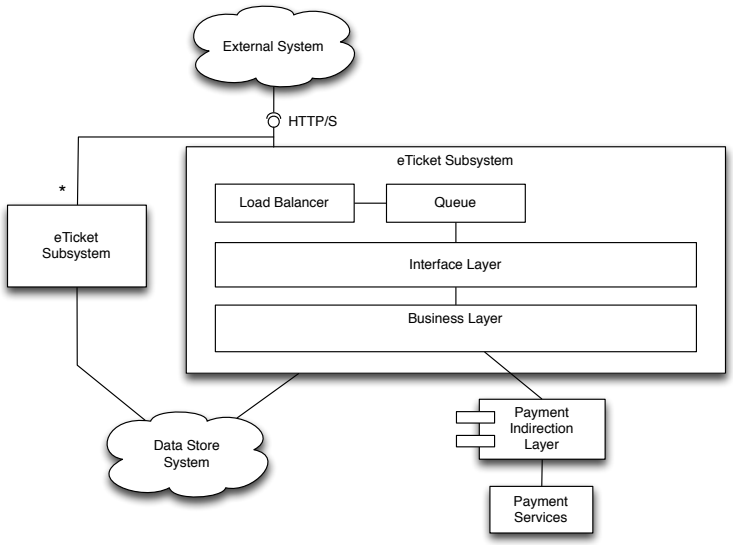

Figure 16: Indirection Layer

Source
Architectural Patterns Revisited – A Pattern Language [**?**]

Issue
The individual payment methods have to be accessed by one or more components, but direct access to these systems is undesirable. We need a way to obfuscate these payment methods.
In the future more payment-methods might have to be added, if all methods are intertwined with multiple components then implementing new methods would be problematic.

Assumptions/Constraints

Positions

1. Indirection Layer
2. Plug-In
3. Interceptor

Decision
The indirection layer wraps up the subsystems, in our case the payment methods. So that the client (our application logic) can make general approaches to the system.

Argument

1. The indirection layer adds wrappers, which helps changeability.
2. The Plugin pattern is used for changes at runtime, which is not something a webservice needs.
3. The Interceptor pattern provides a mechanism for transparently updating the services offered by the framework in response to incoming events. We do not want these changes to payment to be transparent.

Implications
By implementing this pattern, we obfuscate the individual payment methods. This means that for each new payment method, a wrapper must be written and edited. For some payment methods that are not as complex, this might seem like redundancy to the programmer. However we also make sure that all payment is done at one place, making it easier to provide security.

Related Requirements
F-3.1, F3.2, F3.3, RELIAB-5

4.2 Risks Analysis

The table in Appendix A1 shows the risks with their probability (P), severity (S) and weight (W). Probability in this assessment is the chance that the risk happens. Severity describes how severe it is when the risk will happen. The probability and severity have a range from 1 to 4, where 1 is low and 4 is high. The weight is the probability multiplied by its severity.

5 System Architecture

In this section we elaborate on the basic system setup, primarily from a physical point of view. From a very basic point of view the whole system consists of two parts. A simplified physical overview is shown in the image below.

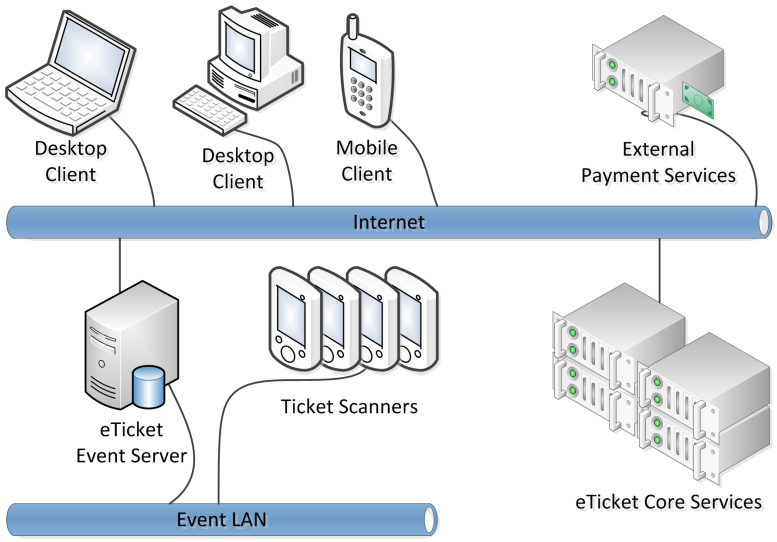

Figure 17: Basic (Physical) View

Core services
System users and event servers connect through the Internet to the eTicket servers. As the amount of service requests is expected to grow in time, distributing the system over multiple servers is inevitable in order to keep it responding quickly. By doing so we will also keep its reliability and security high. For this the system is split into six services, which we explain in the next subsection.

Event services
At each event a temporary server is placed, containing a local copy of the sold tickets' data. Ticket scanners communicate with the (on site) event server for verification of scanned tickets. When a ticket is scanned and verified, it is registered as used in order to prevent multiple persons passing the gate by using the same ticket. The tickets registered as used are synchronized with the main server(s) by the event server, using the Internet connection, for the purpose of providing near-realtime statistics to event organizers.

5.1 Core services

For the sake of performance, reliability and security, eTicket Services are split into six subsystems.

Performance: multiple servers can be load balanced to handle more requests at the same time.

Reliability: a service on multiple servers can failover from a servers hardware failing, keeping the system alive.

Security: possible security breaches are less threatening, as the servers and services can be 'isolated' better.

The 'loose coupling' of components also results in a quite scalable system, but as scalability is not a goal itself - just a way to meet our key-drivers - we will not elaborate on it.

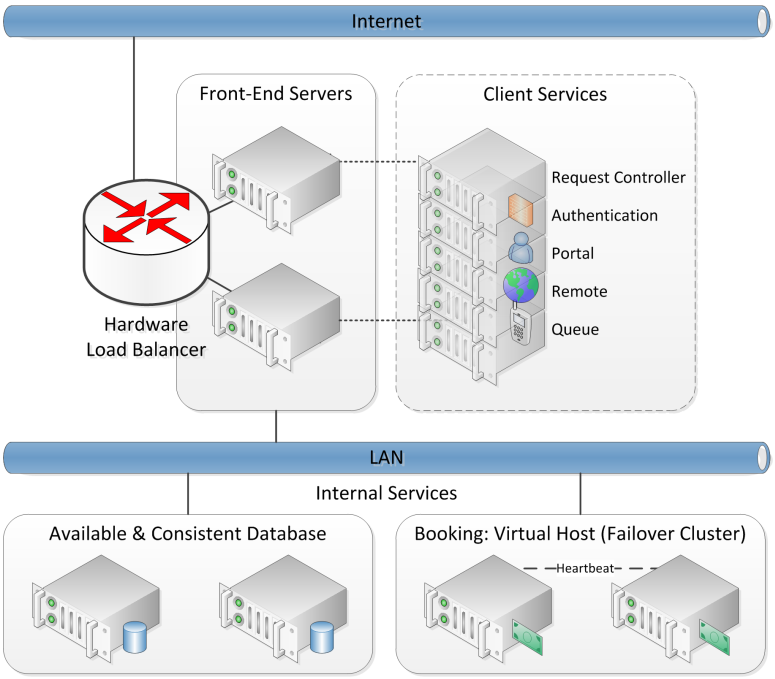

Figure 18: Core Services & Servers

Figure 18 shows a more elaborated model of the subsystems and their positions. This model exists of two significantly different parts: client services, which are optimized for performance and reliable services, that are optimized for reliability.

5.1.1 Client services

All services a client can directly access (through the request controller) are load balanced, including the request handler itself. This way the services are fast, reliable and can be scaled up/out and down. Instead of adding extra complexity to the system by implementing load-balancing ourselves, a hardware load-balancer will be used.
In the initial system these services can run as virtual machines (for every service) on at least 2 virtualization (Front-End) servers, as Figure 18 suggests. This separation of (virtual) machines is necessary to prevent a security breach in one service to threaten other services. If eTicket is willing to invest more money in hardware in order to scale the system up, this can be done easily by transferring the services to separate physical machines. We elaborate on this in the evolution.

Request Controller
: The first subsystem all clients of any service encounter is the request controller. When a client connects to eTicket.com, the request controller redirects him/her to a server that should process his/her request. According to the context in which a client connects, he is redirected to a corresponding service.

Portal
: The portal exists of a Content Management System for serving the eTicket.com web site that displays event information. It is accessible to anyone and also responsible for taking care of simple embedded components, like ticket purchase buttons, on the websites of event-organizers through a separate URL. When an user wants to purchase a ticket, view his profile (personal data, ticket history) or create an event, he/she has to log in. For the login the user is redirected to the authentication subsystem.

Authentication
: Before any client is allowed to access any secured content he/she/it is required to authenticate. The authentication subsystem handles both user and event server log-in requests. If a user is able to provide the right credentials, he receives a security token and is directed to the next service. If an user wanted to see his profile, he is directed back to the portal. In case he wants to purchase a ticket, he is sent to the Queue subsystem. Registration for new users is also made available by this subsystem. The event server is directed to the Remote subsystem after successful authentication.

Remote
: The event server performs its synchronization through this subsystem. The remote subsystem verifies the security token of the event server. According to the identification of the server, access is provided to the part of the database containing data of the ongoing event.

Queue
: The queue protects the booking subsystem, which is used for ticket purchase handling. The amount of concurrent users that can access the booking subsystem is limited by holding the users in a queue. The booking subsystem is made accessible only through the queue subsystem, therefore direct attacks on the booking system are prevented: its resources are not directly accessible to clients and therefore protected from the outside. Furthermore the queue subsystem is able to temporarily block user profile write access from the portal when the queue is full.

5.1.2 Internal services

Internal services are designed with a focus on security, reliability and consistency as their tasks are critical. Client services are expected to minimize and prioritize any requests to the internal services. All internal services implement additional security measures, in order to prevent sensitive data to leak to the outside. Also they cannot be approached directly by clients: interaction with internal services is only made possible through client services.

Booking

This system provides the booking and payment of tickets. It also serves graphically modified versions of the payment wizard, which is part of the embedded component used by event organizers.

Booking is a separate component in Figure 18 as it uses the failover cluster instead of the load-balanced cluster pattern. This requires machines running the service to be configured in a different way. These machines require state replication and a heartbeat link, in order for the take-over to happen successfully.

As purchasing tickets is a critical job and must be done properly under any circumstances, the amount of concurrent users to this subsystem is limited. The transactions have to be written consistently to the database and are also verified by the Booking system, despite the assumption of data consistency in the Data storage subsystem.

Data storage

Common data is stored within a cluster of data storage servers. These servers will run some database software that is designed for availability and consistency (in order to enhance reliability). This database will use data sharding to replicate and distribute its data over multiple physical machines. The connection between the data storage and other servers is typically realized through a Local Area Network, though this can also happen through a VPN if the database is cloned on/devided over multiple physical locations.

Event Server

The event server is a single machine, which is moved from event to event on a regular basis. The machine has software raid capabilities to overcome disk-failures and ECC memory to ensure that no memory errors can occur. The event server has a ethernet lan port to communicate over the network.

A generator can be provided in case power loss could be a concern.

6 Software Architecture

In the following Section the initial model of the eTicket System is illustrated. It shows its subsystem, important components and patterns. A prior version of the initial model is illustrated in Appendix A2.

6.1 Initial Model

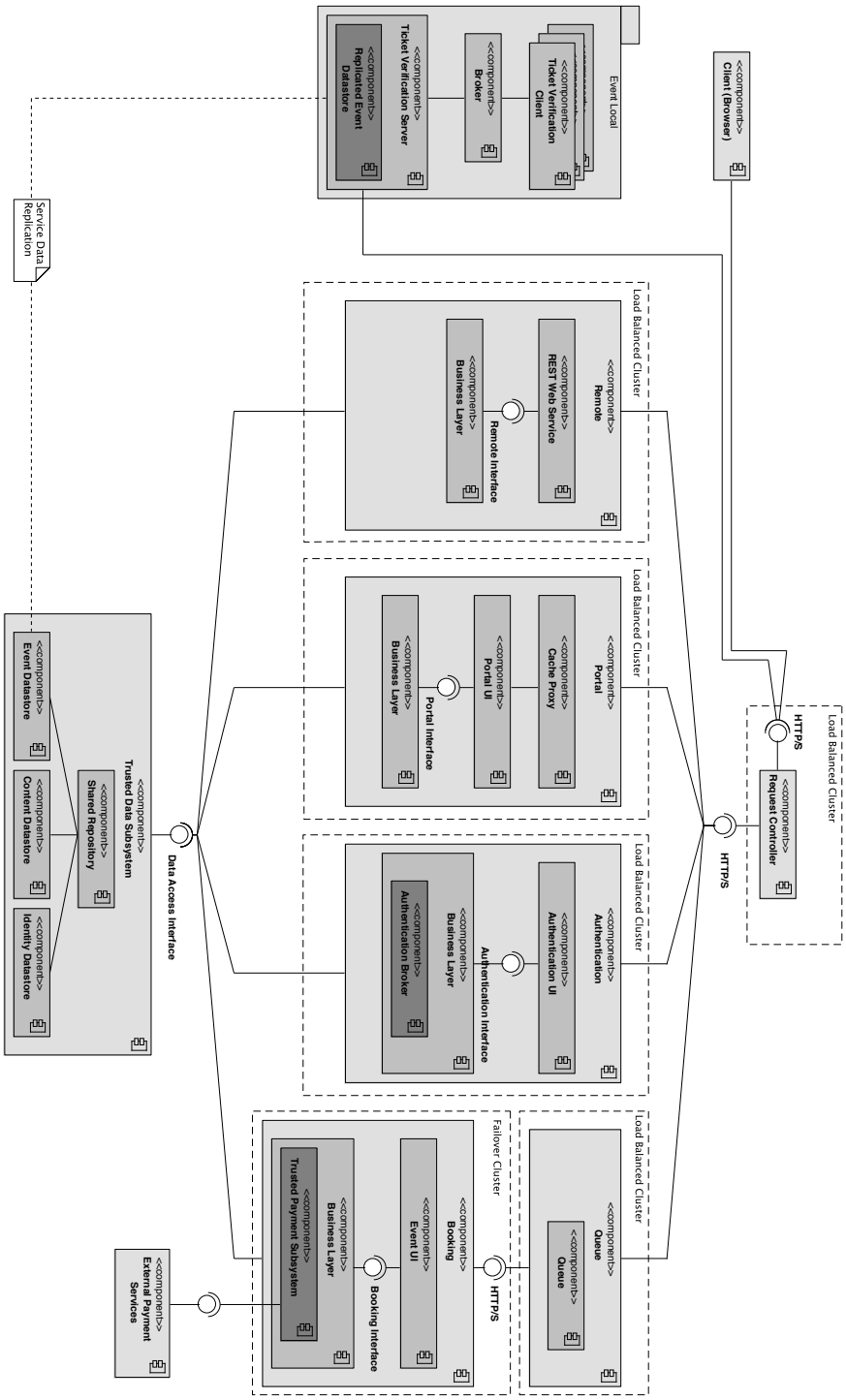

Figure 19: Initial System

6.2 Entire System

In the following section, each subsystem is further described especially with respect to their impact on the key-drivers.

6.2.1 Request Controller

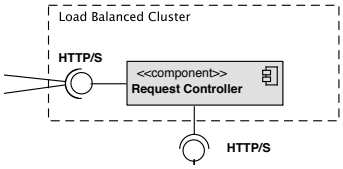

Figure 20: Overview Request Controller

The RequestController is responsible for dispatching incoming HTTP or HTTP/S requests to the correct service. The RequestController uses the information encapsulated in the HTTP-Request to identify the recipient of a request. For instance, the URL can be used to identify the service, as illustrated in Figure 21.

As illustrated in Figure 20 the RequestController provides an HTTP/S interface in order to receive HTTP-Request and it also has an HTTP/S connector in order to forward request to the appropriate service.
Since the RequestController dispatches all incoming requests it needs to be able to handle huge amounts of users simultaneously, therefore the RequestController is replicated by using a Load-Balanced Cluster.

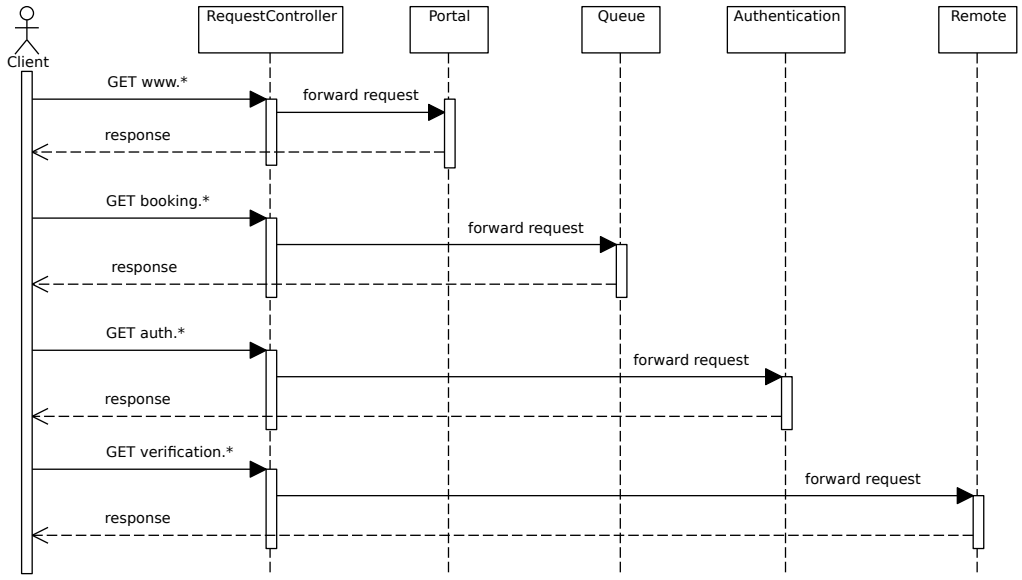

Figure 21: Dispatching request to different services

Security The RequestController is a single point of attack, that can be compromised. Although it does not do complicated business logic, it has to be ensured that it actually forwards request to the correct subsystems.

Reliability The RequestController is a single point of failure, therefore it is required to be replicated via a load balanced cluster. If one RequestHandler fails, requests are forwarded to the other Handler.

Performance All request of the system go through the RequestHandler, hence it is a bottleneck. By replicating this subsystem via a LoadBalanced Cluster the performance bottleneck is addressed.

6.2.2 Portal

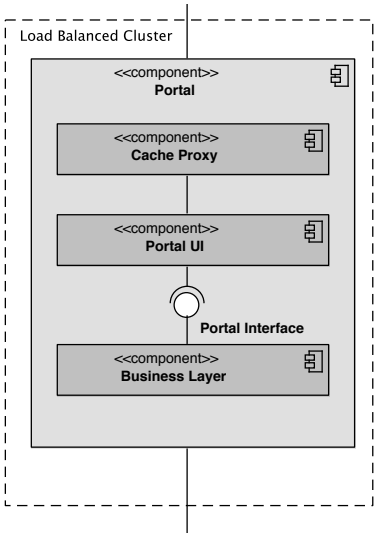

Figure 22: Overview Portal Component

Portal is responsible for displaying the main views of the eTicket system, like home page, event information page etc. It is decomposed into layers. As depicted in Figure 22 the first layer is the cache proxy along with the portal UI and the second layer is the business. The third layer is the data access which is used by all subsystems.

Portal is one of the most accessed subsystems by the users which presupposes the need of handling a considerable number of requests. Therefore a load balanced cluster is used providing replication.

Cache proxy is used because most of the views contain static content. It stores mostly used static content in order to be available upon a request, without the need of querying the content datastore. Portal UI is the layer responsible for displaying the user interface by using a model view controller. The Business layer is responsible for processing the requests forwarded by portal UI.

Security The Portal needs to be secure because it contains components for registration and account management. The structuring into Layers increases security because additional security measurements can be implemented on each layer.

Reliability By replicating the Portal subsystem, the availability of the system is increased.

Performance The structuring into Layers and the communication with the Shared Repository is a performance impediment, however the Portal has a lot of static content and thus is predestinated for the usage of a cache.

6.2.3 Booking

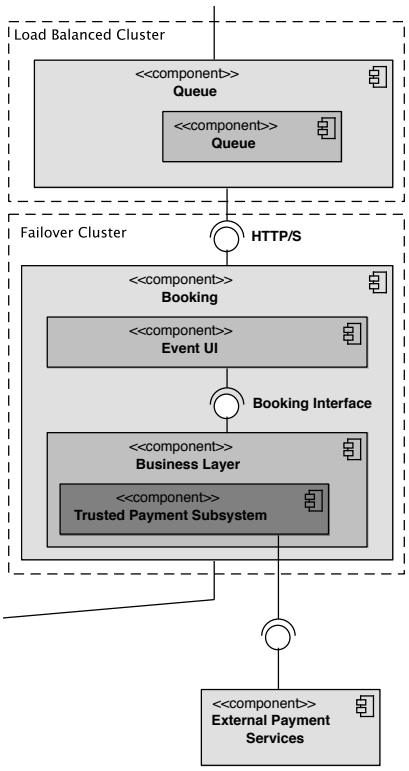

Figure 23: Overview Booking Components

The actual Purchasing is handled by the component illustrated in Figure 23. It consists of two major parts, which are the Queue and the Booking subsystem. Both subsystems handle HTTP/S requests and therefore realize the HTTP/S interface.

According to the non-functional requirement PERFOR-2, the system needs to be able to handle peak loads of users. Therefore, all requests are buffered in a queue before they are forwarded to the actual Booking page. This reduces the load of the system that handles the Booking Process. Additionally, the buffer only queues requests of authenticated users, as illustrated in Figure 24. If a user is not authenticated, he will be redirected to the Authentication subsystem.

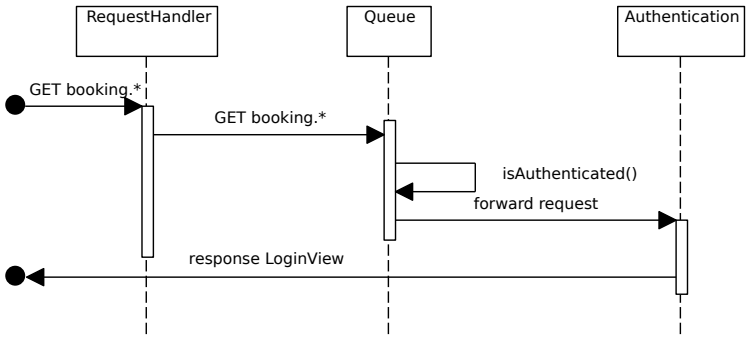

Figure 24: Queue requires Authenticated Users

In case the user is authenticated, an user identification objet will be pushed onto the queue and a temporary user interface will be shown to the user, which shows his position in the queue and an approximation of the waiting time.
As soon as the user identification object is removed from the queue, the request will not be blocked but forwarded to the Booking user-interface. The diagram 25 shows an additional request, which is sent to the Server but it would also be possible to automatically forward the user as soon as he leaves the queue.

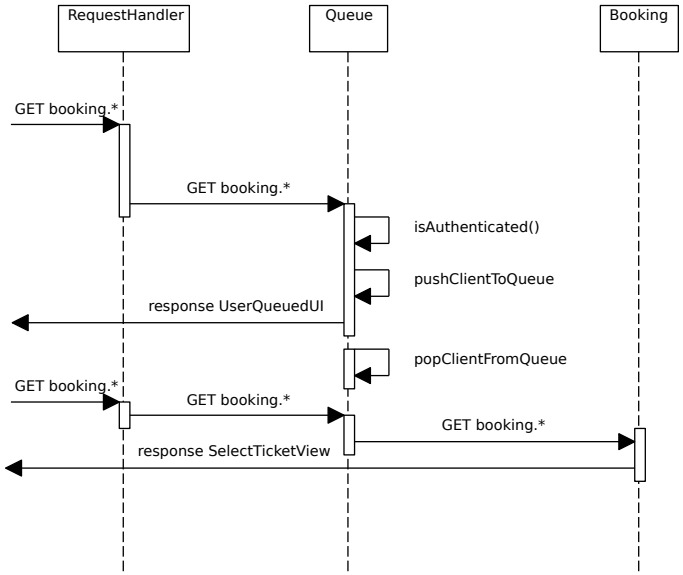

Figure 25: Queuing users before forwarding requests.

The Booking component itself is decomposed into layers. The Event UI Layer provides the user-interface and decouples UI and functionality by using the Model-View-Controller Pattern (cf. Section 6.3.3). The Business Layer contains the Business Processes like Seat Reservation or Payment Processing (cf. Section 6.3.1). Due to the fact that Payment Processing involves handling of sensitive user data and that it is also considered to be interesting for hacking attacks, the Trusted Subsystem pattern is used.
Furthermore, it should be possible to easily adapt to new payment providers in the future. Therefore, the complete payment processing and communication with the external servers is encapsulated in an indirection layer.
The Booking component uses the Shared Repository as datastore for Event and Ticket Data and to share information with other components (e.g. Authentication)

Security The Booking subsystem is critical in terms of security because it deals with money. Therefore it uses a trusted subsystem to increase security when it comes to payment.

Reliability The queue has to cope with huge number of requests, which makes it necessary to evenly distribute requests among a cluster of Queue. If a queue server fails, another server can take over. In contrast, the booking component cannot be easily scaled out, because it has a lot of stateful information like sessions and transactions. In order to increase availability and reliability a failover cluster is used.

Performance Performance is critical for this component. Transaction and the simultaneous access of shared resources like the database and external payment services are considered to be expensive. A plain scale-out approach is not considered to be successful. Instead of increasing the amount of load that the system is able to handle, the load is prevented to occur at the Booking subsystem. Therefore the Queue Buffer is prepended to the actual Booking Process.

6.2.4 Authentication

The authentication component is responsible to provide the login functionality of the system. This component is rather small in size and performs a very isolated task. The subsystem is structured into Layers. The Authentication UI Layer provides the user-interface, which mainly consists of a Login View. The Business Layer contains the logic to validate and verify the user credentials. The Authentication Broker assigns a Token to a user and stores it in the database. For every future request the user provides the token, which is then verified against the token in the database.

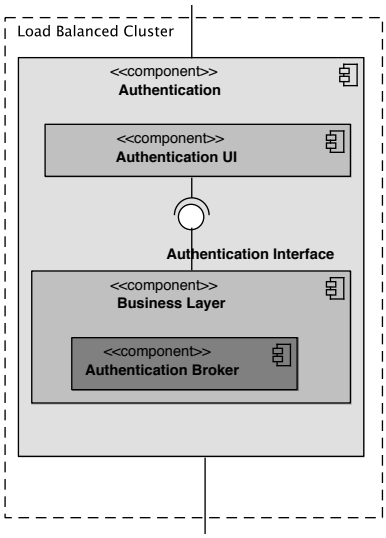

Figure 26: Overview Authentication

Security Encapsulating authentication in a central service increases security because it does not depend on side-effects of other components. Is has to be ensured that the Tokens Store cannot be compromise.

Reliability The authentication component is load-balanced in order to increase availability. Token are stored in the shared repository which is responsible for consistency.

Performance Load balancing increases the throughput of the authentication component.

6.2.5 Ticket Verification

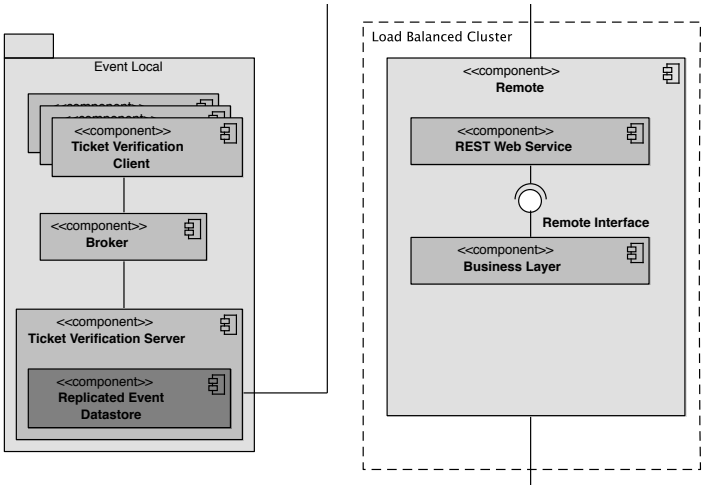

Figure 27: Overview Ticket Verification Components

Figure 27 illustrates components related to the ticket verification procedure. Event local consists of components related to ticket verification processes. Service data replication is used for replicating data related to its event, from the event datastore. The replication will be realized a fixed time before the start of each event. The replicated datastore is hosted on the ticket verification server which communicates with the eTicket system. This communication is realized through internet.

Ticket verification clients are used for the access control, verifying tickets QR codes. Broker is responsible for the communication between ticket verification clients and ticket verification servers. It marshals the data at the sender side and after that forwards it to the receiver side, where the data will be invoked. When a client scans a tickets QR code it sends a request to server in order to verify the QR code and send a response.

The eTicket system provides a REST web service that ticket verification servers use for accessing the business layer services of remote component. A load balanced cluster is used for the remote component. Remote component is structured into layers.

These layers consist of the REST web service, the business layer and the data access layer which is used by all subsystems. In general business layer is responsible for processing the requests from ticket verification servers. It is also responsible for forwarding these requests to data access layer.

Security The authentication mechanism in business layer takes advantage of the layers structure increasing the security. Making use of the broker pattern between the local server and the ticker checker devices enhances the security of the verification part of the system through marshaling data.

Reliability Reliability is increased by the use of the load balanced cluster. The local server augments the reliability of the system as all ticket checker device requests from one event are dedicated to one local server.

Performance The use of layers impedes performance as well as the use of broker because marshaling takes extra time. However because the ticket verification is realized locally by the use of service data replication the performance significantly increases.

6.2.6 Datastore

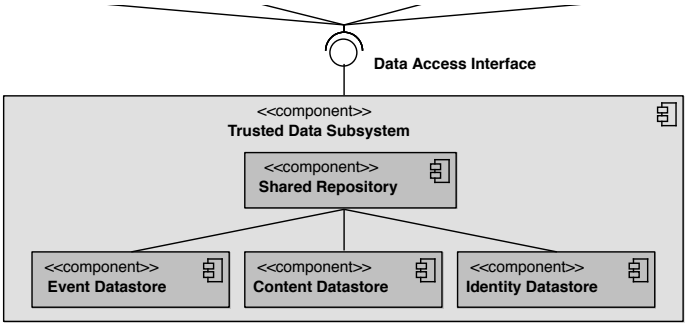

Figure 28: Overview Datastore

This part of the eTicket system is used as the lowest layer of all subsystems through the data access interface, as illustrated in Figure 28. It contains the main data stores of the system which contain event, content and identity data.

Trusted data subsystem is responsible for enhancing the security of the datastore. For every request forwarded to the datastore, it creates new security tokens. These tokens are used during the internal process in the datastore. Upon completion of the process trusted data subsystem responds to the request.

Shared repository is used in order to provide an API for components of subsystems to access the main data stores. It is mainly used due to the multiple type of data stores, to provide additional security as well as that for the need that many clients need to access the data stores.

Security The use of layers in combination with trusted data subsystem significantly increases the security of the system. Shared repository provides additional security.

Reliability The shared repository is replicated to increase availability, which requires additional effort to keep data consistent.

Performance Shared repository as well as trusted data subsystem impede performance. The latter due to the use of extra authentication mechanisms.

6.3 Elaborated Model with Patterns

In this section an elaborated model of the system with patterns will be given. Each pattern is described in detail concerning its actual implementation as well as its effect in the key drivers.

6.3.1 Layers Pattern

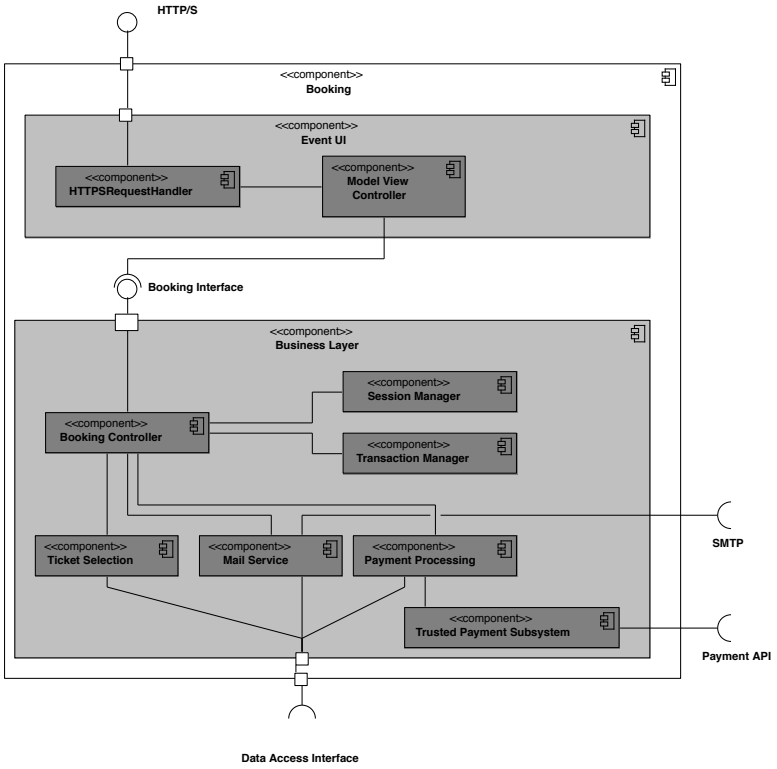

Figure 29: eTicket Service Layers

The Layers pattern is used for the four subsystems (Remote, Authentication, Portal and Booking) in the eTicket System. The systems are generally decomposed into two layers. The first is responsible for providing a user or machine interface and the second layer encapsulates the business logic (a third layer is the data layer,which is not illustrated in the initial model). In general the use of layers facilitate security in each layer, however it would decrease performance due to additional indirections.

In the following, the concrete decomposition of the Booking subsystem into Layers is illustrated (cf. Figure29). A description of the decomposition of the other subsystem is omitted, because of the similarity.

The Event UI Layer provides a HTTPS connector in order to receive the client requests. These requests are processed by the HTTPSRequestHandler component, which dispatches it to a concrete Controller within the Model View Controller component (cf. Section 6.3.3). The MVC component uses the Booking Interface of the Business Layer to invoke the actual purchasing process.

The Booking Controller ensures that each client follows the correct process of purchasing tickets. Therefore it communicates with the Session Manager, which is responsible for maintaining Session Data, and the Transaction Manager, which is responsible for handling Transactions within this Subsystem e.g. Start, Commit and Rollback of a Transaction.

The Mail Service component is responsible for sending the purchased Tickets via E-Mail to the

customer. Therefore it connects to Mail Servers via the SMTP Protocol. The Payment Processing uses the Trusted Payment Subsystem, which is an Indirection Layer. In order to perform a payment transaction, the request must pass through payment trust subsystem (Section 6.3.4) which is responsible for securing those processes and communicating with the external payment services through payment plug-ins component (cf. 6.3.4).

6.3.2 Brokered Authentication Pattern

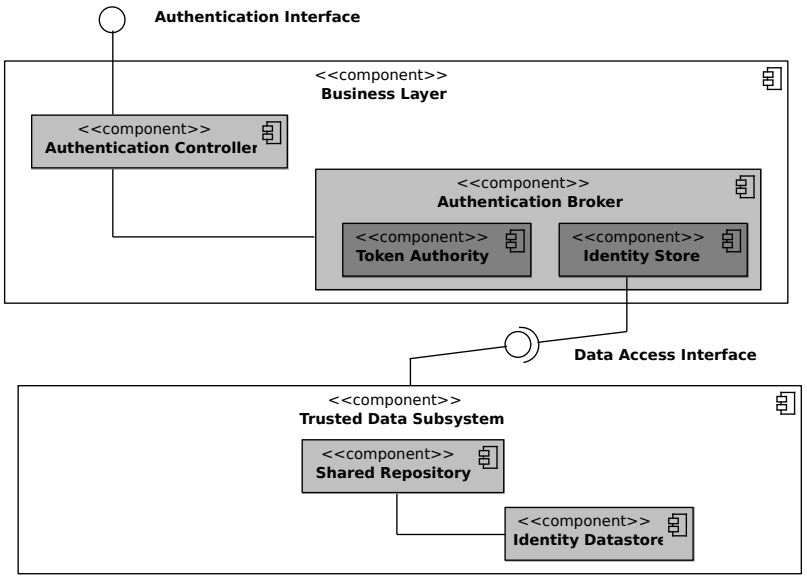

Figure 30: Brokered Authentication on Interface layer

A client[5] needs to be authenticated in case he want to access certain functionality of the system like Purchasing Tickets, Changing Account details or Creating Events. Therefore a clients request is forwarded to the Authentication subsystem. After the client provides the login credentials, the Authentication Controller is invoked through the Authentication Interface.

The Authentication Controller controls the business process of validating the credentials and verifying them against the database. It invokes the Authentication Broker. The Authentication Broker asks the Token Authority to create a new Token for the session of the client. The Token is associated with the clients identity and stored into the Shared Repository. Additionally, the Token is returned to the client, which need to show it every time he access a protected subsystem. The Token could be a certificate, which is stored in the certificate store of the Browser or the Token could be stored as a Cookie.

[5]A client could be a humans or a machine.

6.3.3 Model-View-Controller Pattern

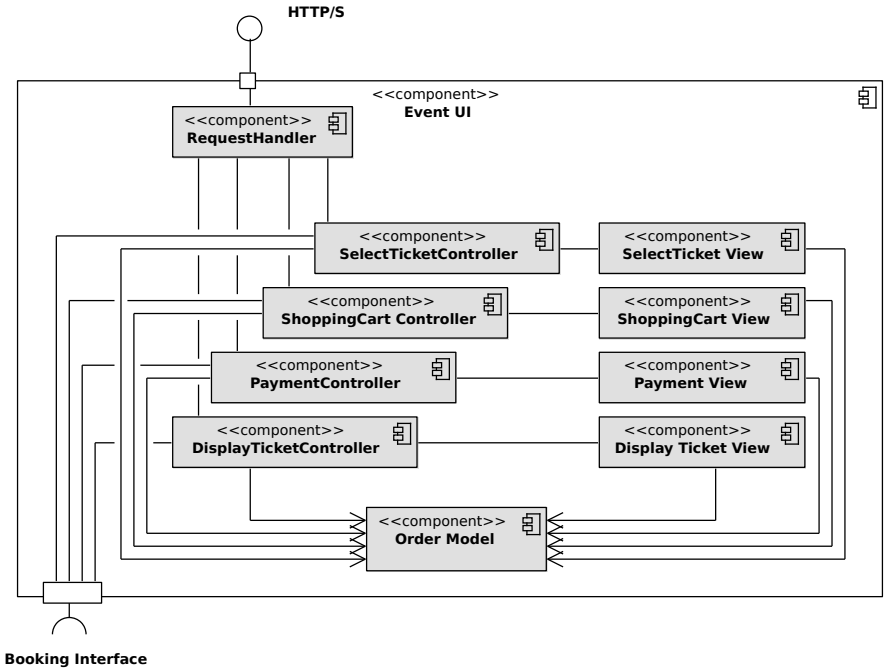

Figure 31: Model-View-Controller decomposition in the Booking UI Layer

The Model-View-Controller is used to decouple user-interfaces and functionality from each other. The Pattern has been used in the UI Layer of the Booking subsystem and the Portal subsystem, because these are the Subsystem that have a complex user-interface with a lot of functionality.

Figure 31 exemplifies a possible decomposition of the Booking UI Layer. The Request Handler receives an HTTPS request and invokes a particular Controller, which modifies the Model and responds with a particular View. The Booking Process is especially suited for the MVC pattern because it shows a lot of different Views of the same Model, in this case the OrderModel. There are even more Views involved, which are not shown in the diagram for reasons of simplicity.

6.3.4 Payment Trusted Subsystem Pattern

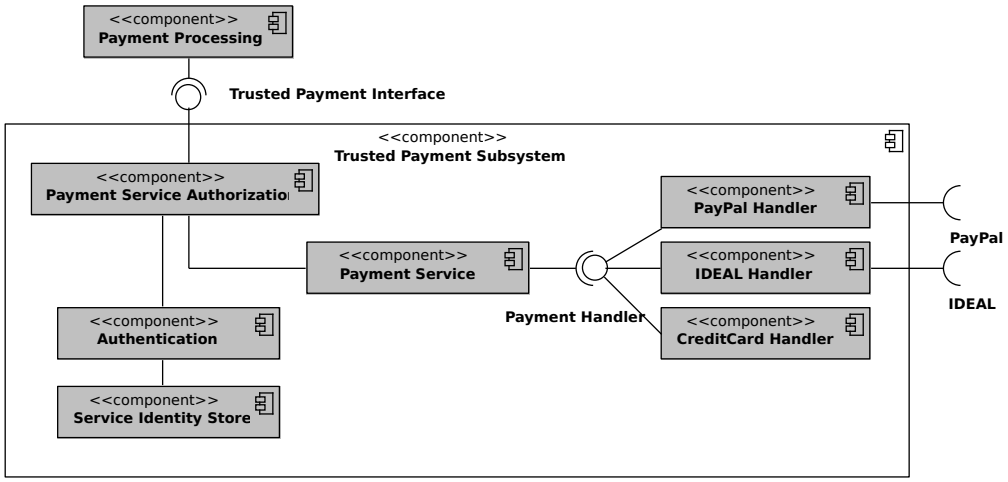

Figure 32: Payment Trusted Subsystem on Business layer of Booking Component

After ticket booking completes, a payment transaction must be realized. As illustrated in Figure 32 trusted payment subsystem is used before any payment transaction takes place further enhancing the security of the system. Payment processing component is responsible for forwarding the requests to trusted payment subsystem.

Trusted payment subsystem prevents the requests from directly using the payment handler components. When these requests reach payment service authorization component, then new credentials are created for this particular service with new security tokens. Authentication and service identity store components are responsible for issuing the new security tokens.

The requests with the service's credentials and security tokens are then forwarded to payment service component. This component is responsible for processing these requests as well as forwarding them to payment handling components so a payment transaction can be completed. Payment handling components realize the communication with the external payment services like PayPal and IDEAL.

In this way only the payment service can use the payment transactions services with it's own security tokens and not client requests which were issued by the brokered authentication pattern.

An implication of this pattern is that it is a single point of entrance. If malicious users exploit it then they gain access to payment transactions processes. However with the use of brokered authentication pattern at the same time, the possibility of this to happen is significantly decreased.

6.3.5 Cache Proxy Pattern

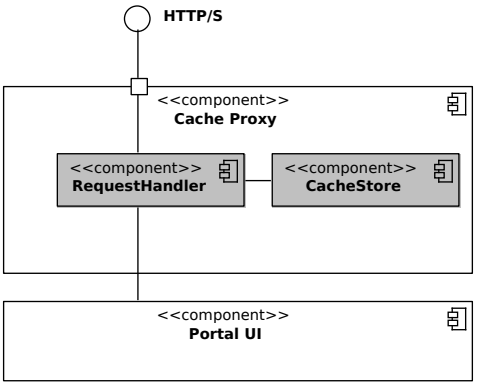

Figure 33: Cache Proxy on Portal Component

Cache proxy is used on Portal component before portal UI. As illustrated in Figure 33 it provides a HTTP/S interface for incoming requests. The request handler of the cache proxy processes all incoming requests. Cache proxy pattern is used in front of the Portal UI in order to temporarily store data that have been requested. In this way there is significant increase in the performance of the system.

Upon the arrival of a request, request handler searches the cache store and if the data are already stored then it serves the request with a response. Otherwise the request is forwarded so data can be retrieved from the main data stores instead. Requests related to processes of storing into the data stores do not get affected by cache proxy.

When a request is served with a response from the main data stores then request handler of cache proxy component stores the data which have been requested. If the same data is requested again , it will be immediately available. Request handler also takes into consideration the available space in the cache store. If cache store is nearly full then older or less used data are deleted.

6.3.6 Broker

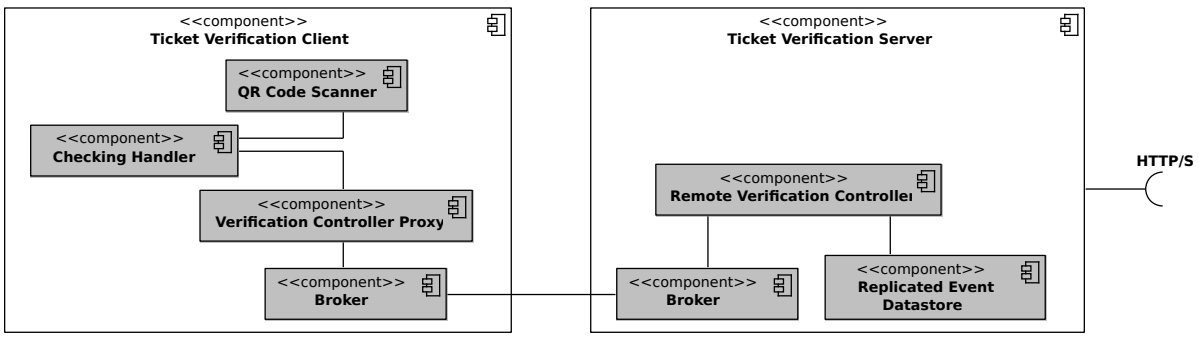

Figure 34: Broker on Event Local

A Client-Server relationship between the ticket verification clients is realized by broker pattern as depicted in Figure 34. The broker pattern ensures safe and reliable communication between the clients and the ticket verification server, enforcing the reliability of the system.

Broker's main responsibility is to act on a verification method on the server by invoking it. Before this happens though, the argument (the QR code) is extracted from the actual ticket by QR code scanner and checking handler. Then through verification controller proxy is forwarded to client's broker where it is marshaled and forwarded across to server's broker.
After that the remote verification controller invokes the object specified (verification method), and unmarshals the argument (QR code). Finally the verification method is executed comparing the scanned QR code and the stored ticket data in the replicated event data store. After that a positive or negative response is sent back.

By using the broker, ticket verification is effectively realized at the ticket verification server. The consequence of this is that there are no consistency issues, and therefore security is enforced. Namely a person can't use the same ticket twice, or hand over a duplicate ticket to be used by another person for entering the event at the same time.

6.3.7 Service Data Replication

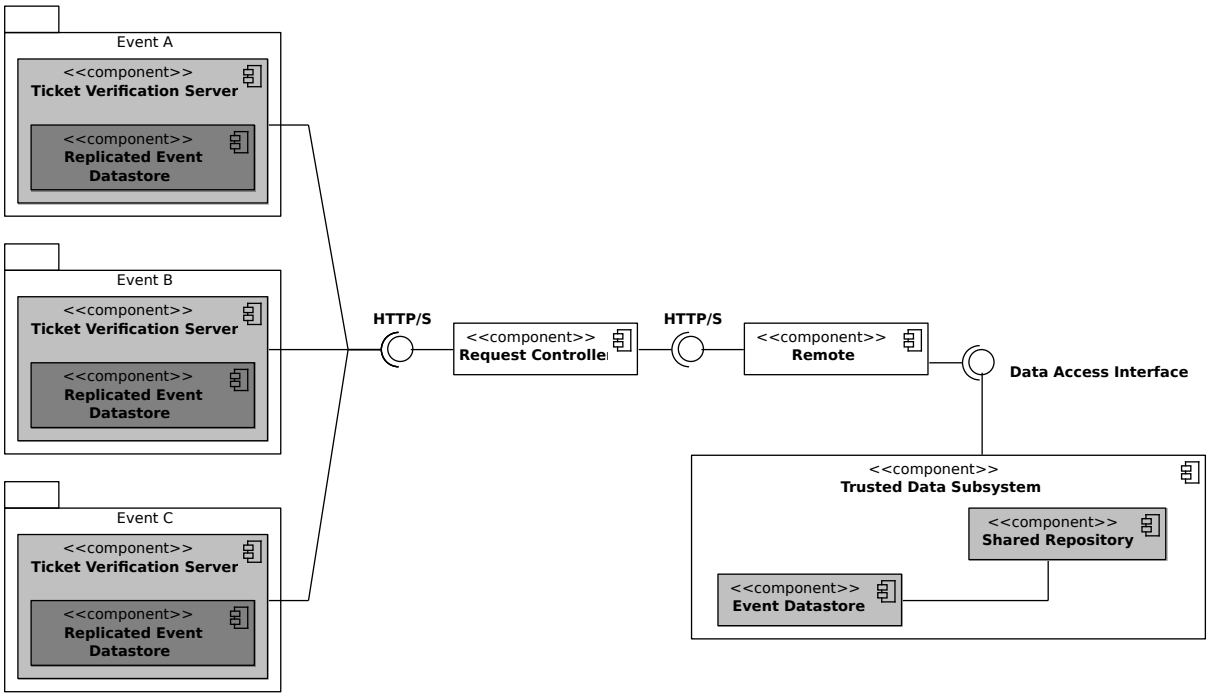

Figure 35: Service Data Replication on Event Local

Service data replication pattern is used for creating replicas of data related to each event from the event data store. As illustrated in Figure 35 ticket verification servers communicate through HTTP/S with the request controller and hence with the main system.

A fixed time before the start of the event and after the ticket booking has been disabled for that particular event, a replica of data related to it is created from the event data store. Ticket verification servers sends a request, which is forwarded to the remote component through the request

handler. Then the request is forwarded again to the trusted data subsystem which returns a response with the requested data.

Using service data replication, the performance of the system related to ticket verification processes is significantly increased. Additionally due to none dependency on the internet availability, the reliability is also increased.

7 Evaluation

7.1 Key-Driver Validation

In this section we will check if the patterns used support our key drivers. Our key drivers are:

- Security
- Reliability
- Performance

The design is evaluated through the use of force resolution maps. We will first will make a FRM for each pattern we used. Then we will make a FRM for each partition of our system as we partitioned our system for the PDAP[6] process.

7.1.1 Layers

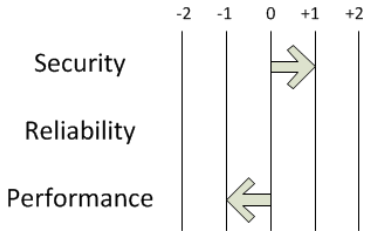

Figure 36: Layers force resolution map

Layers provide an interface for access to each layer. This increases the security of our system since we can monitor incoming data at that single point of entry and nulify possible attacks. Performance is hindered though because the majority of the requests have to go through all the layers to be served. This overhead decreases the performance of our system.

7.1.2 Shared Repository

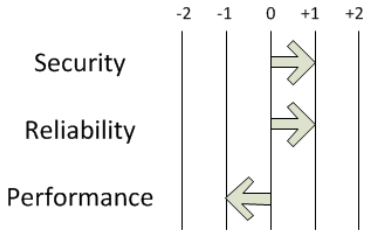

Figure 37: Shared repository force resolution map

The shared repository pattern increases the security of our system because it only provides a single point of access. This way we have a central point where we can monitor the incoming requests. The reliability of the system is also increased because the shared repository because data consistency is enforced. Performance is hindered though because the single point of entry can become a bottleneck if our system gets too many requests.

[6]Pattern-Driven Architectural Partitioning

7.1.3 Model-View-Controller

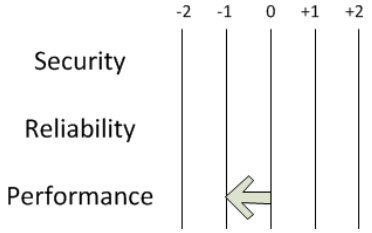

Figure 38: Model view controller force resolution map

Model view controller hinders the performance of our system due to the increased overhead from notifications. On the other hand it increases the maintenability of our system which is not a key driver, but is an important quality attribute as well.

7.1.4 Broker

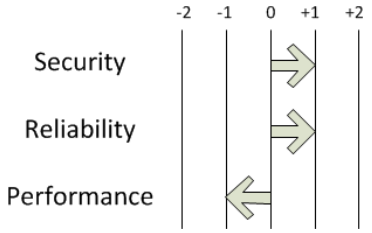

Figure 39: Broker force resolution map

The broker pattern increases the reliability of our system by ensuring that messages reach their destination. It also increases the security of our system by the use of marshalling. The performance is hindered though because messages are not directly sent to the reciever but have to go through a process before. This overhead has a negative impact on performance.

7.1.5 Service Data Replication

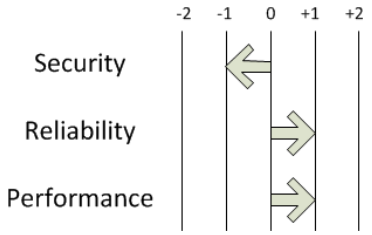

Figure 40: Service data replication force resolution map

Service data replication increases the reliability of our system by making sure that there is no dependency on the internet availability. We transfer the data to the local event server before the event so there's no need for constant communication with our main servers. The performance is also increased since the ticket checking devices are near the local event server, thus the response times for ticket validation are low. The security of our system is hindered though because it is an external system and it poses an easier target to attack.

7.1.6 Brokered Authentication

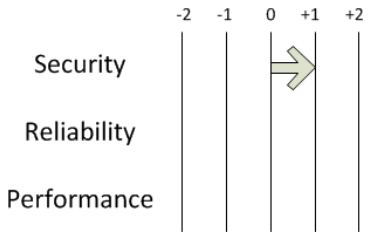

Figure 41: Brokered authentication force resolution map

The brokered authentication increases the security of our system because it is used to ensure that clients are authenticated before they can request a service that requires them to be authenticated.

7.1.7 Trusted Subsystem

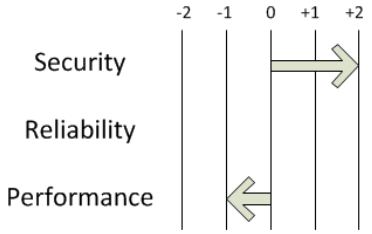

Figure 42: Trusted subsystem force resolution map

The trusted subsystem greatly enchances the security for the areas that have been assigned to be under its supervision. However due to the additional overhead for the additional credential checking, performance is hindered.

7.1.8 Cache Proxy

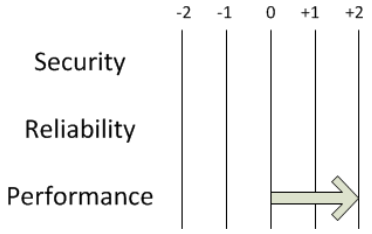

Figure 43: Cache proxy force resolution map

The cache proxy pattern takes a load off the database by caching data that are often requested. This relieves the database from constant transanctions about the same data, thus increasing performance.

7.1.9 Load-Balance Cluster

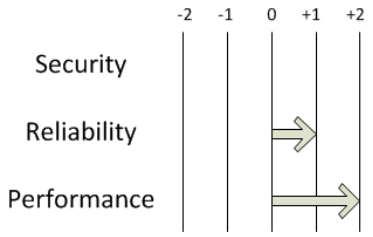

Figure 44: Load-balance cluster force resolution map

The load-balanced cluster pattern increases the availability of our services by allowing the system to be distributed over many servers, which makes our system more reliable. It also greatly increases the performance of our system by making it easy to add more servers and scale the system. By adding more servers we can achieve acceptable performance even when our system is under heavy load.

7.1.10 Failover Cluster

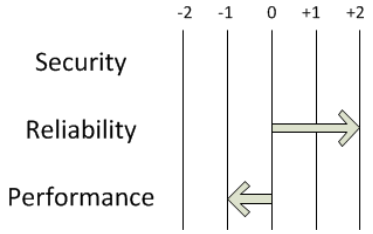

Figure 45: Failover cluster force resolution map

The failover cluster greatly increases the reliability of our system. By splitting a service over many different servers and sharing state between them, failure of a server will not cause the particular service to become unresponsive. It will continue with its operation normally. It also hinders performance due to the extra overhead for syncing states between the different machines.

7.2 Subsystems

Our system is divided into six different subsystems which are mentioned in Section 6.2. We are going to evaluate each subsystem on the following subsections.

7.2.1 Request controller subsystem

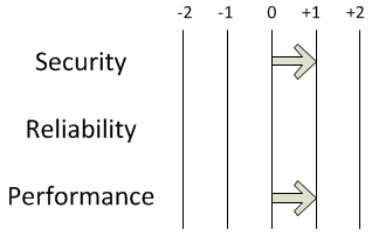

Figure 46: Request controller subsystem force resolution map

The request controller subsystem provides a single point of entry which is good for the security of our system, because we can easily monitor the incoming requests. It is also a single point of failure which is bad for the reliability of our system. To counter that problem we used the load-balanced cluster pattern to replicate it. Because it handles all the incoming requests it can easily become a bottleneck. But the load-balanced cluster pattern helps us cope with that issue. By replicating this subsystem we can ensure that it will have good performance even under heavy load.

7.2.2 Portal subsystem

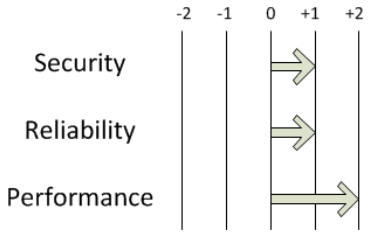

Figure 47: Portal subsystem force resolution map

Four patterns are present on this subsystem. They are model view controller, cache proxy, load-balance cluster and layers. The model view controller is used to display the web pages to the users. As we noted earlier model view controller increases the maintenability which is not a key driver and it hinders the performance because of the additional overhead from notifications. The cache proxy greatly increases the performance of this subsystem because most requests will be served by the cache proxy instead of going through all the layers to the datastore subsystem. The load-balance cluster increases the reliability and greatly enchances the performance of this subsystem. Last the layers increase the security and hinder the performance. Performance is greatly enchanced by the cache proxy and the load-balance cluster though so performance is not decreased.

7.2.3 Booking subsystem

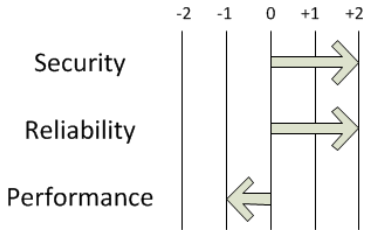

Figure 48: Booking subsystem force resolution map

Five patterns have been used on this subsystem. They are layers, trusted subsystem, load-balance cluster, failover cluster and model view controller. The layers pattern increases the security of our system because it provides a single point of entry. It also reduces the performance of our system because requests must go through all layers to be served. The trusted subsystem is only used in the payment process. It greatly increases the security of the subsystem but it also reduces the performance due to the additional overhead. The load balance cluster pattern increases the reliability and performance of the queue. The queue itself increases the reliability of our system

because we will serve the requests in a controlled rate. It also decreases the usability of our system but that is not a key driver. By handling the customers at a standard rate performance is increased by the queue. Finally the failover cluster greatly increases the reliability of our system by splitting a service over many servers, thus making the failure of a server tolerable. It also descreases performance due to the additional overhead for syncing states between servers. The model view controller also impedes performance due to the additional overhead it generates for serving views.

To sum up we get a +2 for security due to the layers and the secure subsystem patterns and we also get a +2 for reliability due to the use of the queue and the load-balance cluster. Performance is a -1 because of the layers, the trusted subsystem, the failover cluster and the model view controller. The plus performance from the load-balance cluster and the queue even it out and it goes to -1 instead of -2.

7.2.4 Authentication subsystem

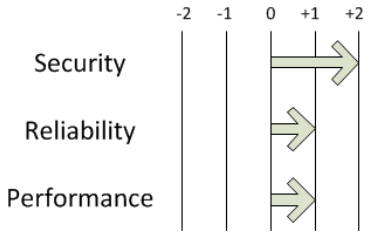

Figure 49: Authentication subsystem force resolution map

Three patterns are on this subsystem, authentication broker, load-balance cluster and layers. The authentication broker increases the security of this subsystem. The load-balance cluster greatly increases the performance of the authentication subsystem. It also increases reliability. Last the layers pattern increases the security of the subsystem but at the same time hinders its performance.

7.2.5 Ticket verification subsystem

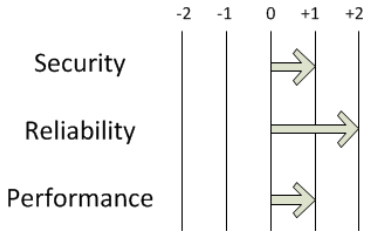

Figure 50: Ticket verification subsystem force resolution map

Four patterns are used in this subsystem namely, broker, service data replication, load-balance cluster and layers. Broker increases the security because the transmitions from the ticket verification devices to the local ticket verification server are being marshaled. Due to marshaling performance is reduced because of the additional overhead for marshaling. Service data replication increases the reliability and the performance of this subsystem. It increases performance because the server is local to the ticket checking devices, thus great speeds can be achieved. Also reliability is increased since there is no dependency on internet for ticket validation. Security is

decreased though by service data replication because it is seperate in a way from our system and it forms an easier target to attack.

The load balance cluster on remote component on our system increases the reliability of this subsystem since it can be replicated. It also has a positive effect on performance again because it can be replicated. Last the layers pattern increases the security of our system because there is a single point of entry and the authentication mechanism on the business layer increases the security. It also impedes performance, but it is already taken care of by the load-balance cluster. To sum up performance is not hindered but it is plus due to the locality of the server and the ticket checking devices. Also performance is not an issue on the remote component because it is load-balanced. Security is hindered a bit due to the use of service data replication but it is stilll plus due to broker and layers. Last reliability is greatly increased because there is no dependency on the internet and due to the load-balance cluster.

7.2.6 Datastore subsystem

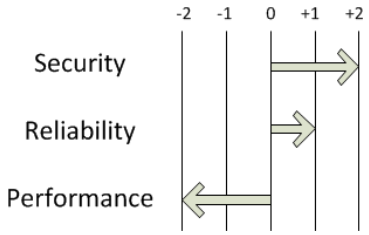

Figure 51: Datastore subsystem force resolution map

Three patterns are present on this subsystem. They are shared repository, secure subsystem and layers. The shared repository increases reliability since it ensures data consistency. The shared repository on the other hand impedes performance because a lot of components connect to it which can become a bottleneck. The shared repository also increases security since it provides a single point of entry thus an easy way to check incoming requests. The use of trusted subsystem also greatly increases the security of this subsystem. It also reduces performance due to the additional overhead for the checking of credentials. Last layers increases the security of the subsystem but reduces the performance due to the single point of entry. The single point of entry disadvantage of shared repository and layers does not add up though.

7.3 Complete Architecture

For the force resolution map of the complete architecture we are going to take into account each subsystem that we evaluated. If we add the force resolution maps for the subsystems we get:

- Security: +9
- Reliability: +7
- Performance: +2

These totals also need to be divided by 6 because we have 6 subsystems. We get +1.5 for security, +1 for reliability and for performance we get a 0. Thus after having argued for each subsystem separately we can say that these are the attributes for the force resolution map of our whole system.

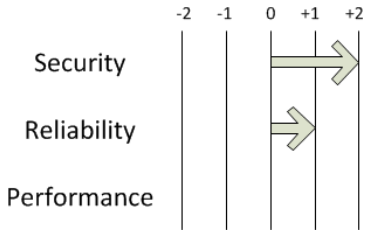

Figure 52: Complete architecture force resolution map

7.4 Requirement Verification

Numbers to Pattern Table

Number	Pattern
1	Layers
2	Model-View-Controller
3	Shared Repository
4	Service Data Replication
5	Broker
6	Brokered Authentication
7	Trusted Subsystem
8	Cache Proxy
9	Load Balancer Cluster
10	Failover Cluster
11	Indirection Layer

7.4.1 Functional

Ticket Sales

Requirement	Pattern	Remarks
F-1.1	6	The brokered authentication pattern verifies the login requirement. The user interface for the login is provided by the authentication UI component in the authentication component. The differentiation of customer and event organizer accounts will be considered during implementation.
F-1.2	2	The components in the business layer of the booking component verify this requirement. The pages are displayed to the user using the MVC pattern.
F-1.3	-	This is verified by the mail service component in the business layer of the booking component. After a client has succesfully completed the transanction, the ticket will be e-mailed to the customer.
F-1.4	-	The components in the business layer of the booking component verify that a seat is locked out for other customers when a seat is booked.

F-1.5	2 ; 3	This requirement is verified by the shared repository pattern, since that information will be stored on the shared repository. The authentication UI component in the authentication component verifies that the user submitting and modifying that data.
F-1.6	2 ; 3 ; 6	This is verified by the shared repository, because it will store the data that associate the purchase of a ticket with a particular customer. The user will have to authenticate in order to access that information, thus the brokered authentication verifies this requirement. The pages for the actual modification are displayed using the MVC pattern which also contributes to the verification of the requirement.

Managing Events

F-2.1	2 ; 3 ; 6	This is verified by the shared repository pattern because the created events are stored there. MVC also verifies this requirement since it provides the user interface at the portal UI component for the event organizer to create or modify events. Also the event organizer needs to go through the brokered authentication to verify that it is his / her event.
F-2.2	2 ; 3	Event information is stored in the shared repository. This is also verified by the use of the MVC pattern in the portal UI component that provide the user interface to provide that information.
F-2.3	2 ; 3	Event information is stored in the shared repository. This is also verified by the use of the MVC pattern in the portal UI component that provide the user interface to provide that information.
F-2.4	2 ; 3	Event information is stored in the shared repository. This is also verified by the use of the MVC pattern in the portal UI component that provide the user interface to provide that information.
F-2.5	2	The MVC will communicate with the business logic to verify if the bank account is verified.
F-2.6	-	Verified by the business logic in the event creating part of the system.
F-2.7	3	This is verified by the shared repository as the generic maps for event locations will be stored there.
F-2.8	2 ; 3 ; 6	First an event manager should verify himself and after that there is the opportunity within the view to store a plan in the shared repository.
F-2.9	2	This is verified by the MVC pattern which provides an easy way to customize web pages.
F-2.10	-	This is verified by the components in the business layer of the booking component.
F-2.11	3	This is verified by the shared repository pattern as it will be used to store those statistics.

F-2.12	3	This is verified by the shared repository as these options will be stored there and they will be associated with the specific event organizer account.

Handling Payment

F-3.1	7 ; 11	This is verified by the payment processing components and the indirection layer in the trusted Payment subsystem of the booking component.
F-3.2	7 ; 11	This is verified by the trusted subsystem pattern which provides a connection to the payment processes via the indirection layer.
F-3.3	7 ; 11	This is verified by indirection layer in the trusted subsystem pattern which provides a secure way to access the payment services that eTicket supports.

Entrance Control

F-4.1	5	The broker pattern is used for communication between the event server and ticket checker devices.
F-4.2	4 ; 5	The broker pattern is used for communication between event and main server. The event server is a replicated one.

7.4.2 Non-Functional

Reliability

RELIAB-1	8 ; 9 ; 10	By using a proxy pattern and a load balancing pattern before reading the pages the load of the system will be more balanced what is will increase the availability of the system.
RELIAB-2	5	Verified by using the Replicated data server pattern for a local server and the main server with a updating interval of 30 seconds. So within 30 seconds of a ticket sold, it should be registered as valid on the local server at the event.
RELIAB-3	4 ; 5	All the bought tickets are available on the event server and can be requested by the ticket checker devices through the broker pattern much faster than when it should go over the internet and through the layers of the main server.
RELIAB-4	-	The components in the business logic layer of the booking component will handle this.
RELIAB-5	7 ; 11	Verified by the trusted payment subsystem pattern, the indirection layer and the components in the business layer of the booking component, where different payment processes are handled.

Security

SECUR-1	1 ; 6 ; 7	Verified by the Brokered Authentication pattern, where customers are verified if they are acknowledged to enter parts of the system. For the payments the trusted payment subsystem prevents any request from directly using the payment handler com- ponents.
SECUR-2	1 ; 3 ; 7	Verified by shared repository where data is securely stored since it is also an area monitored by the data trusted subsystem. The trusted payments subsystem also verifies this requirement since that is where sensitive payment data is processed.

Usability

USAB-1	2	Verified by the development of neat views in the MVC pattern.

Performance

PERFOR-1	2 ; 9 ; 10	Verified by the implementation of the model view controller, the load balancer cluster when a peak load occurs and the failover cluster when servers go down.
PERFOR-2	9	Verified by the load balancing from the load balancer cluster.
PERFOR-3	4 ; 9	Verified by replicating the data from to the event to a local allocated event server.

Scalability

SCALAB-1	8 ; 9	Verified by using the proxy, load balancer cluster and the queue component. The first one will make sure that load on the system by only viewers is handled separately. The load of requests and transaction on the database from customers who want to buy a ticket during ticket booking peaks is balanced according to the second. The queue component also verifies this requirement by ensuring that not all customers try to book a ticket at the same time.

Adaptibility

ADAPT-1	7	Verified by the trusted payments subsystem and the payment service component, where more payment handlers can connect to the payment service component.

8 Evolution

In this section possible evolution steps of the system are discussed. We do not provide a concrete time roadmap according to which the system should be extended, as extenstion of the system is most probable to be driven by the amount of users and events it has to serve. Also there is not a best order in which the opportunities should be implemented, as they all optimize different aspects of the system.

8.1 Upscaling

As the system will grow in time, due to an increasing amount of events and customers, scalability will probably become a key-driver in the future. However, scalability is a result of the current design as explained in section 5, hence the system is easy to scale up already. The initial design (Section 5, Figure 18) shows that all performance services running on virtual machines. In order to scale the system up, additional virtualization servers could be added. Though a way to achieve better results, by reducing indirection, is to locate every service on a physical machine. This is illustrated in the figure below.

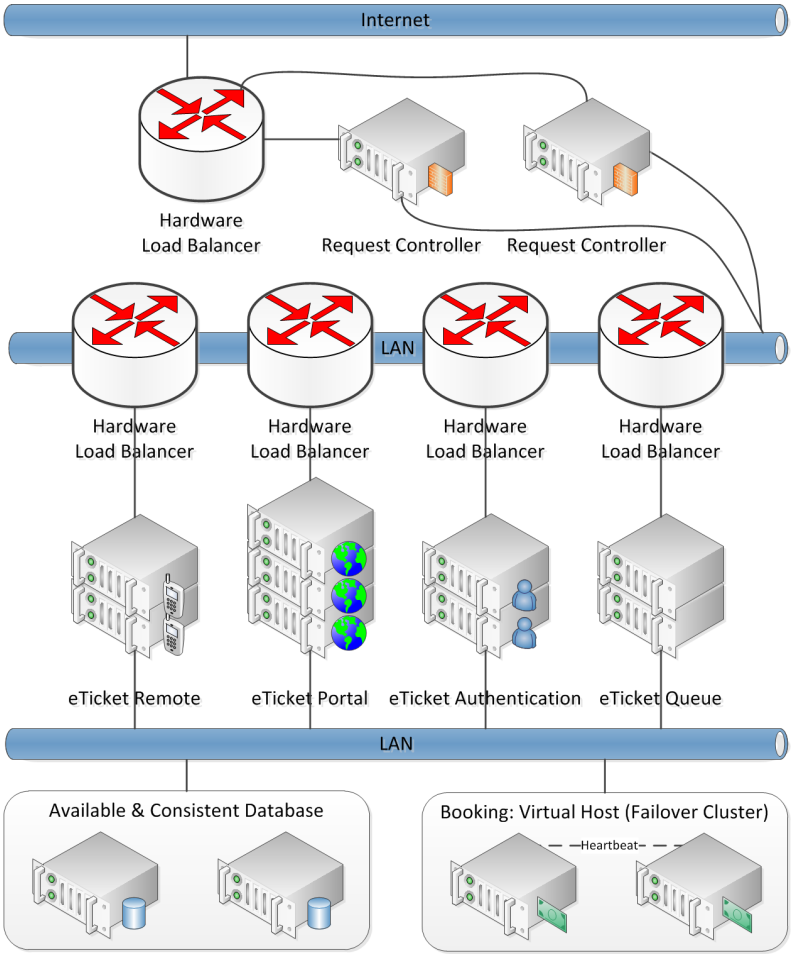

Figure 53: Services tranferred to physical machines

By splitting up the system this way, every performance service can be scaled up separately, enabling targeted performance and/or reliability improvements. The Portal servers are likely to be utilized most, as users often want to check the web-site without actually purchasing tickets and/or performing operations that require Authentication.

8.2 Geographical Distribution

w Toegether with the popularity of eTicket the geographical coverage is expected to grow. From this the necessity will arise to locate servers on different geographical locations. If the servers are located in different data centers, the clients will have to be directed to the closest server in order to gain the best performance. By using managed DNS services, also known as DNS load balancing, this is trivial to accomplish.

The computational power from different locations can be combined by connecting the locations using a Virtual Private Network. For example: when some large event is created in Groningen, attracting a lot of users at the same time, connections to servers can be dispatched to Amsterdam when the ones in Groningen become highly loaded. Due to the systems scalable design, all performance services can be scaled out easily. This is visualized in the image below.

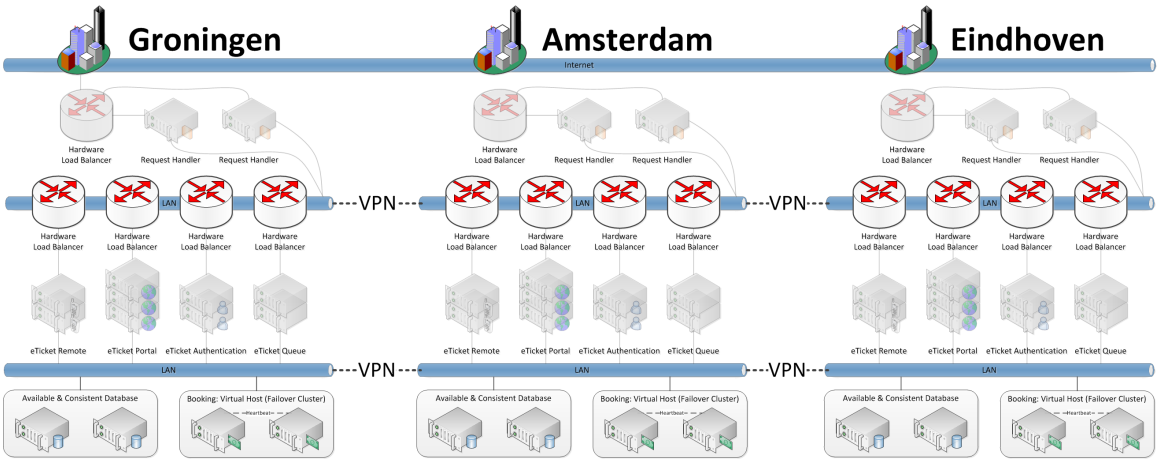

Figure 54: Geographical Distribution

To achieve such distribution, it is sufficient to instruct the hardware load-balancers to cooperate with their friend load-balancers through VPN. As hardware supporting this feature might be expensive, it seems wise to overthink this before purchasing the initial hardware, in order to prevent the need for replacing it afterwards.

Interconnecting the different geographical locations also renders the opportunity to store the event data at the relevant location, e.g. the data for an event at Schiphol is stored in Asmterdam. For this the sharded database proves useful, as the locations of data shards can be configured at data-store level; the data is then written to the right shard. Other services do not care where the data is coming from, as long as they get the data. Furthermore, at this point the datastores can be configured for eventual consistency, under the assumption that the 'eventual consistent' stores are remote and only used for the sake of backing up, rather than reading.

8.3 On demand scaling

When a lot of servers are used, chances are great the utilization of the machines will be low most of the time. Costs of energy and data consumption can raise high without the machines being useful at all. This only makes sense for physical machines, as these machines can be put in stand-by mode when the load is low and waken up when the load rises, reducing the power consumption drastically. An example scenario is shown below.

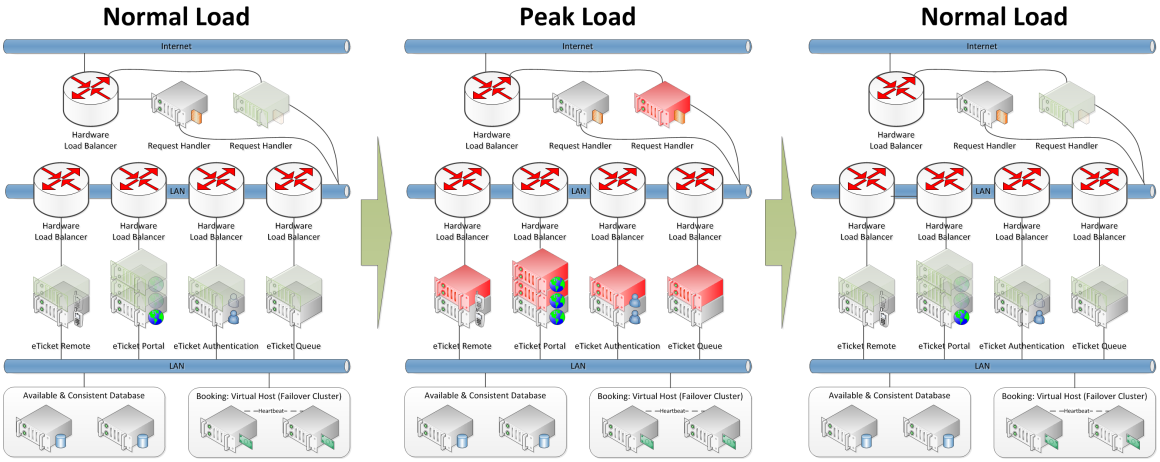

Figure 55: On demand capacity scaling. Green: temporary stand-by, Red: temporary powered-on, Grey: always on.

On demand capacity scaling can be added to the system at any time, though it is most useful when eTicket has many physical servers (on multiple locations). On demand capacity is primarily applicable to the peformance services. The technique is less suitable for reliable services, as these have to be reliable and accessible at any time.

9 Conclusion

The first section will first re-evaluate the completeness of our interpretation of the Pattern-Driven Architectural Partitioning process. After this evaluation a more interesting section will explain the larger concerns that we still have.

9.1 PDAP Process

This document has followed the Pattern-Driven Architectural Partitioning process as goal to architecturally describe a system using patterns.
The next few paragraphs give a quick overview about how we applied the PDAP process to the eTicket process.

Identify architectural drivers Step The first step in this process is to identify the most prominent architectural drivers of the system. In the system overview, we have identified the key drivers by prioritizing functional and non-functional quality attributes according to stakeholder concerns.

The resulting key drivers after careful consideration of the stakeholders concerns were identified as:

- Security
- Reliability
- Performance

Candidate architecture pattern selection Step

In chapter four we described the final patterns along with decisions and rationals, as well as alternatives and why we did not choose them. Selecting candidate patterns to address the needs of the architectural drivers was a difficult task. We decided to use multiple pattern languages when the Architectural Patterns Revisited - document fell short. This means that some patterns complement each other more than others, and for example, the service data replication pattern serves as a high-order pattern.
The patterns are listed in a chronological order, meaning they continually build up the components/qualities that are needed to match the system to our key drivers and requirements.

Partitioning Step

During the creation of the initial model in Chapter 6.1 and during the analysis phase of our project we tried to piece the patterns together, to get the best possible decomposition for our problems. As can be seen in the section we decided on two large partitions. Namely ticket verification and distribution. Where distribution is partitioned into it's own components. We found that this partition scheme has allowed us to more quickly think about what patterns to use and how they should work as a whole.

Evaluation Step

The evaluation chapter contains evaluations regarding the different forces of the different patterns. We found that the best way to express this evaluation would be through Force Resolution Maps. A relevant evaluation concerning the key drivers and the used patterns can also be found there.

PDAP Thoughts

PDAP is of course a pattern to be used with iteration. The final version as can be seen now is the result of many iterations where constantly new patterns or old patterns were discarded as they did not fit together, or did not make sense. Sometimes patterns were not architecturally relevant and only helped in creating more paperwork were also removed from the document to make the design more lean and comprehendible.
Our group is in agreement, that by focussing on patterns, the wheel does not constantly have to be re-invented. And problems and solutions to these problems come more natural.

9.2 Final Word

The key drivers Performance and Reliability (namely consistency) are perhaps the most conflicting of key-drivers. In our efforts to mitigate each others forces we rely heavily on the Failover and Load balancer patterns to both increase performance and reliability.
Also we had to deal with the large number of incidental increases of load. To solve this we rely on the Queue server, where users would be stuck if the server is handling more than x requests. We believe that this queue will work, although in reality this is no sure-fire solution.
However the queue is needed to offload the database. We have actually toyed with the idea to cut off normal user-logins and profile changes etc in the presentation layer if load would increase too much, to allow the database to do more transactions concerning payments/buying tickets which is why there is such a heavy load.

We thank you for reading this document, and we as a group especially enjoyed the heavy debates we had regarding our project. In summary we found this particular project to be very educational.

Appendix

A1 Risk Analysis

Risk Categorie	P	S	W	Responsibility	Threshold	Prevention	Reaction
Business							
Unable to deliver a qualitative product within the time given.	1	4	4	Development team	The deadline and costs estimate are exceeded.	Regular communication within the group about the progress of parts related to their agreed milestones	Because there is no possibility to increase the amount of resources, the resources should increase their amount of hours spending on the project.
Technology							
Main Server Crashes	1	3	3	e-Ticket	The main server doesn't respond as should be.	The main server is duplicated. More instances are available.	When one server crashes another instance can take over.
Event Server Crashes	1	4	4	eTicket	The ticket checker devices can't reach the event server	Making use of highly reliable servers which are used on location of an event.	Restart the server and connect to the main server to gather all the data needed for an event.
Ticket Checker Device Crashes	1	2	2	eTicket	The ticket checker device can't validate any ticket.	Ticket checker devices have little	Every event has multiple ticket checker devices. When one crashes another device can be used.
Internet connection unstable during event	2	2	4	Event Organizer	The local event server can't reach the main server.	The event server has a copy of the event data from the main server and connects every 30 seconds for updates. It doesn't need to connect to the main server for all ticket validations.	When connection is lost, it tries to connect again in 30 seconds.
Peak loads for event booking	4	1	4	e-Ticket	The load of requests for ticket booking reaches 6000 per second.	The load is balanced in the design of the system.	The system queues the customers and let them wait until they can be serviced in a responsible manner.

Peak loads for ticket validation for concurrent events.	4	1	4	e-Ticket	Just before an event starts a lot of customers want to enter the venue.	Every event has its allocated server where all the specific event requests are handled separately.	Event validation requests are handled by their local event server.
Hackers try to acquire sensitive customer data.	2	4	8	e-Ticket	An unauthorized visitor acquired access to the system.	Authorization is needed to acquire access to system parts.	Analyze the vulnerability of the system and update the system to prevent further hacks.
Third party payment handler component can't be reached.	2	4	8	Third Party payment handlers	A payment transaction can't be closed.	Make use of reliable payment processes.	The system supports more options for payment. If one fails it can try another.
Process							
Missing important requirments	1	3	3	Software Architects	Stakeholders are not satisfied with system	Communicate with stakeholders closely as early as possible	Communicate with the specific complaining stakeholder(s).
Other							
Significant risks haven't been identified	1	2	2	Software Architects	An unexpected Risk occurs	Short and quick iterations.	Best effort to deal with the occurred risk.

A2 First Approach

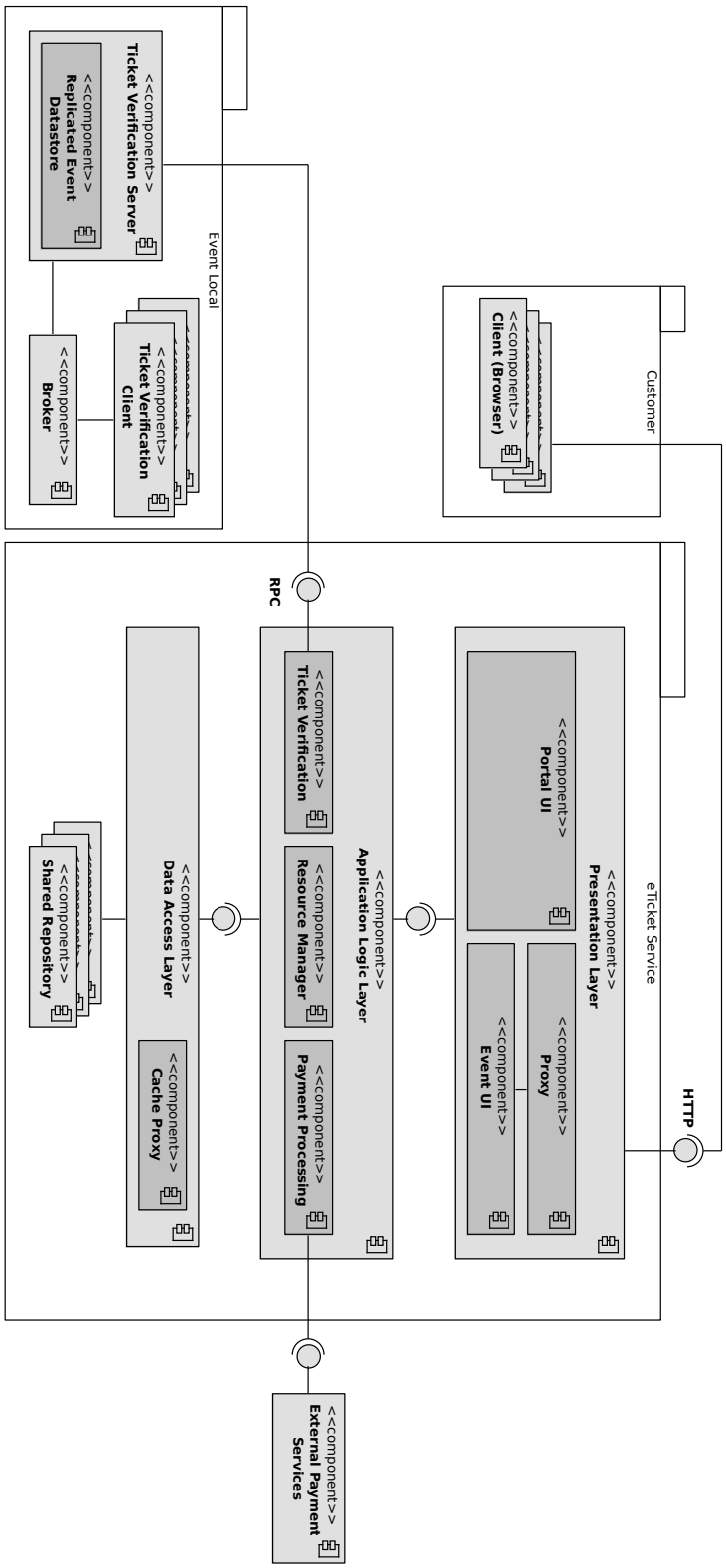

Figure 56: First Approach of the Initial Model

A3 Time Tracking

Week 1

Person	Task	Hours
cm	SVN Setup, Document template	2
	Chapter outline and document decomposition	1
		3
si	Introduction on business model, System Context	2
		2
jh	Installing and getting to know SVN and TeXworks	2
		2
wb	Created Collaboration tool (etherpad)	1.30
	Fixed introduction	1
		2.30
fk	Draft of introduction	1
		1
ah		
		0
	Group total	**10.30**

Week 2

Person	Task	Hours
cm	Reading PDAP	1
	Review Meeting	1
	Meeting	2
	Improv Introduction	2
	Improv System Overview	4
	Product function description - payment	1
	Include Review Notes, Changed outline, Prepared Requirements	1
		12
si	Creating meeting agenda	1.5
	Reading PDAP	1
	Meeting	2
	HL-1, HL-4, functional and non-functional reqs	2.5
	Patterns reading	2
	Architectural Decisions - Patterns designing	4.5
		13.5
jh	Reading PDAP	1
	Coach Meeting	1
	Team Meeting	3
	System Overview	4
		9
wb	Review Meeting	1
	Meeting	3
	Getting to know this damn latex layout thing	0.5
		4.5
fk	Reading PDAP	1
	Review Meeting	1
	Meeting	2
	HL-2, HL-3, functional reqs	2
	Read updated document and go through some typos	0.5
		6.5
ah	Meeting with coach + team	1
	Read assignment + present documents	0.75
	Team meeting	2
	Write notes + update hours	0.5
	Work on stakeholders, waste loads of time on latex tables	2
	Finish stakeholders + reformulate system context	2
		8.25
	Group total	**53.75**

Week 3

Person	Task	Hours
cm	Coach meeting	1
	Group Meetings	5
	Pattern MVC	1
		7
si	Coach meeting	1
	Group Meetings	5
	Meeting agenda	1
	Patterns reading and designing	6
		13
jh	Reading Doc + Preparing Meeting	0.5
	Coach Meeting	1
	Group Meetings	5
	Research + Adjusting System Overview	2
	Research + Creating Pattern	3
		11.5
wb	Coach meeting	1
	Group meeting	2
	Group meeting	3
	Intro, rereading and changes	0.5
	Shared repository Pattern research	0.5
	Shared repository Pattern document	1.5
	Client Server	1
	Diagramming which was fruitless..	0.2
	General document scouring and cleanup	1
	final review,check	1
		11.2
fk	Coach meeting	1
	Group meeting	2
	Group meeting	3
	Seat planning and requirements	1.5
	Read about the proxy pattern	0.75
	Research and work on chapter 4 proxy pattern	2
	Read about RLM pattern	0.75
	Research abd work on chapter 4 RLM pattern	1.5
	Redo diagrams for CH4	1
	Research on patterns	2
	Reworked proxy and RLM for CH4	1.5
		17
ah	Coach meeting	1
	Group meeting	2
	Group meeting	3
	Rework stakeholders + key-drivers	2.5
	Review + improve section 1,2	2
		10.5
	Group total	**70.2**

Week 4

Person	Task	Hours
cm	Coach meetings	1
	Team meetings	1.5
	Team meetings	5
	Team meetings	1
	Diagrams Entire system and elaborated model	12
	Elaborated model	4
	Review and general Improvements	4.4
	Entire system	3
		31.5
si	Patterns reading and designing	8
	Group 1 review	2.5
	Team and coach meeting	6.25
	Entire system design	3.5
	Elaborated model design	24.5
		44.75
jh	Meeting Coach	1
	Team Meeting	3
	Presentation	1.5
	Adjusting Requirements	0.5
	Updating old patterns	1
	Research Patterns	2
	RPC pattern	1
	Plugin	1
	6.2.2	2
	6.3	1.5
	Requirements Verfication	3
	Correspondence	4
	Risk Analysis	3
	Requirements Verification	3
	Updating patterns	1.5
	Reviewing	1.5
	Last minute changes	2.5
		33
wb	Team and coach meeting	2.5
	Patterns Indirection, Shared Repository,Broker,service data replication	8
	Fixed all patterns to be consistent and clear	5
	Reviewed entire document	2
	Conclusion	3
	Entire System, ticket verification	2
	General document	3
	Rewrite of Requirements	2.5
	Group meeting	5
	Last minute changes	2
		35
fk	Team and coach meeting	2.5
	Read group 1 document	1.5
	Prepare presentation	2
	Review stakeholders and key drivers	1
	Review and restructure patterns	2
	Change RLM to asynchronous queue	1.5
	Research on patterns	2
	Write about cache proxy pattern	1

	Group meeting	5
	Draft of FRM	1.5
	Work on FRM	1.5
	Group meeting	1
	Work on evaluation chapter	4.5
	Finish FRM section	1
	Read document and general fixes	2.5
		34.5
ah	Review + improvements sections 3,4	1.75
	Team + coach meeting	2.5
	Presentation draft	1
	Correspondence about presentation, system model	0.75
	Rework initial models + presentation	3.5
	Correspondence about project	1
	System architecture + skype discussion	3.5
	Evolution	3.5
	Functional Requirement Evaluation	1
	Rework & discuss system architecture + evolution diagrams	4.5
	Team meeting	0.75
	Patterns + rework system architecture + evolution diagrams	8.5
	Rework evolution text + diagram	2
	Finish evolution	1
	Review chapter 6,7, fix minor things chapter 5,8	2
		42.25
	Group total	**221**