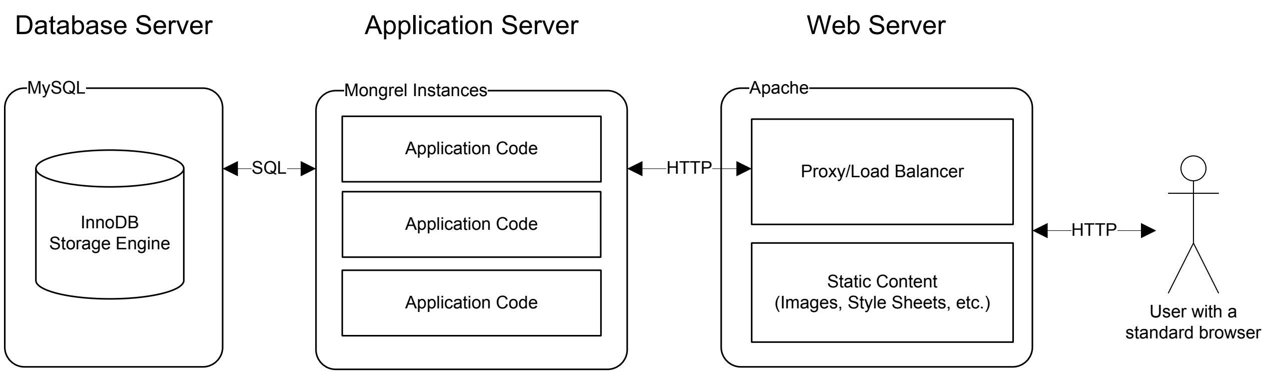
Account This!

Group 9

Kristoffer Renholm
Johannes Edelstam
Joakim Ekberg
Jesper Skoglund

2.2 Overall Architecture Description

The overall system architecture is divided into three major components, the database, the application and the web server. Each component has its own responsibility and is communicating with other components through standardized protocols and communication channels.



The web server serves as the front-end towards the user. It is responsible for serving the user's request for static content like pictures and style sheets, and forwarding of requests to the Application Server, that will generate a dynamic response or allow user input.

The application server is responsible for running the Account This! codebase. Due to lack of threading in most Ruby applications there is no natural way of processing and responding to concurrent requests from users. Therefore multiple instances of the special hosting server software for Ruby on Rails applications called Mongrel will be run concurrently on different TCP/IP ports. The web server is responsible for distributing the load over these instances even.

The database server is responsible for persisting data between user sessions. The database server uses a transactional storage engine with support for relations between tables. This reduces the risk for corrupt data.

2.3 Detailed Architecture

2.3.1 Model View Controller

Model View Controller (MVC) is a well known software design pattern. Its purpose is like many other design patterns to organize the code in a maintainable way.

The MVC principle divides the application into three separate subsystems called layers. Model, View and the Controller layer described as follow:

- *Model* - The representation of the domain specific entries that builds up foundation of the application. This could be anything from users, shopping charts, accounts and so on.
- *View* - The views purpose is to, as the name hints, present the data in different ways.
- *Controller* - The controller represents the glue in the application. The controller layer directs traffic inside the application, all from querying the models for information as rendering views to the end user.

When building according to the MVC pattern, program code is separated in different layers. The code never floats around, ie, the design pattern makes every line of code to live in one of the three layers.

2.3.2 MVC implementation in Ruby on Rails

This section of the document is to describe how Ruby On Rails (Rails) implements the Model View Controller (MVC) design pattern.

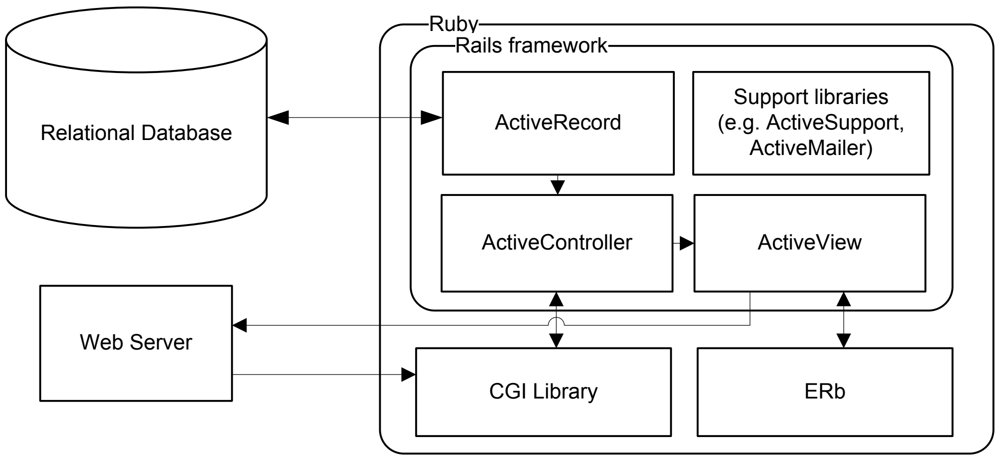
When writing code in the MVC design pattern each line of code fits into one of the three layers of the application, model, view or controller.

In accordance with its MVC foundations, Rails is made up of three different subsystems. Separate in each sense that they could be used individually. These are:

MVC Phase	Rails subsystem	Purpose
Model	ActiveRecord	The model provides the bridge between the database and the Ruby code that builds up the models. ActiveRecord provides several functions to read, manipulate, find, etc. to data. One important quality of ActiveRecord that it generates Ruby-methods from each field in a database column. Therefore it's very easy to translate the database to a set of model objects.
View	ActionView	The views of a Rails project usually contains of several XHTML documents with embedded ruby to display the dynamic content for the specific view.
Controller	ActionController	The ActionController represents the final glue needed to connect the view with the model. It handles forms and determine which view to render according to the data that the user inputs.

What is Rails then? We now know that it's built up from three components that have different purpose according to the MVC pattern.

Rails is the only necessary infrastructure that is needed to connect these three different subsystems together to create a great web framework.



2.3.3 The file structure of a Ruby on Rails project

The file structure consists of more or less important directories. They are predefined by Ruby on Rails. This section describes them and ho the files inside them are named.

app/	Holds most of the source code. All the project specific code is placed here.
app/controllers/	Here all controllers are placed. They are named like vouchers_controller.rb so that the paths will be recognized automatically by Rails.
app/models/	Here all models are placed. They will be named as voucher.rb
app/views/	Holds all of the template files that will be rendered from the controllers, or by another template. They are named like vouchers/show.html.erb. Where 'html' could be any format and show is the name of the corresponding action for show in the vouchers controller. A template could also be named _voucher.html.erb. Then it is a so called partial, which often are used to e.g. render a row in a table describing a voucher. A file could also be named as vouchers/index.xml.builder, that means that it is a template which generates XML that corresponds to the vouchers controller index action.
app/views/layouts/	Here all layouts are stored. A layout is a template which is used in most cases for the whole application. Then it will be named application.html.erb. It will contain html that will be rendered for each request (e.g. menus and such). Somewhere in the layout a yield will be made to render the actual template inside the layout.
app/helpers/	Holds all helpers for the project. A helper could be named as vouchers.rb. It will contain methods that the voucher template uses. There will also be a helper named application.rb which will contain methods available to all templates.
config/	This directory is very much described by it's name. It holds all configuration files, such as routes.rb which contains information about how rails should create pathes. It also contains databases.yml which describes all databases that the project is using. Also all configuration files for the environments are kept here.
components/	This directory was used in previous versions of Rails, but not any more.
db/	Contains the auto generated file schema.rb which describes how the database tables where created.
db/migrate/	This directory holds all migration files. They describe how all the tables should be created. They are named as 009_create_vouchers.rb.
doc/	Auto generated documentation for the project will end up here.
lib/	Any extensions or classes that isn't models or controllers ends up here. It could i.e. be a parser or similar.
public/	This directory holds all files that should be directly available from a web browser. Like images, javascripts and CSS style sheets.
script/	Everything in this directory is auto generated when the project is created. It is different scripts to e.g. generate new controllers and models, or to destroy them and all the files that belongs to them.
test/	Testing is important to every large software project. The test/ directory contains all functional and unit tests along with fixtures. Fixtures are files to load test data in the databases.
vendor/	Holds libraries that the project rely on. It also contains the plugins directory which holds all plugins used by the project.

2.3.4 RESTful development

Account This! will be written with the quite new technique called RESTful Rails. REST is short for Representational State Transfer. The concept is to take advantage from that the HTTP protocol standard uses more than just POST, and GET. It also uses the methods PUT and DELETE. Every URL should map to a resource on which you can perform any of these methods instead of URLs that maps to certain actions. E.g. the URL /vouchers/ can typically be called with the GET method that would correspond to get all vouchers. A POST to the very same URL would instead correspond to creating a new voucher. The new voucher would be accessed e.g. at the URL /vouchers/4. A call with the method PUT to that URL would result in an update of that specific voucher. A call with the method DELETE would result in destroying that voucher. RESTful also handles different formats in a very clever way. Since every URL is an URL to a resource every URL would theoretically be called for any format. Like /vouchers.xml and /vouchers/4.xml. That would then result in getting the output from the voucher controller as XML.

HTTP Verb	REST-URL	Action	URL without REST
GET	/vouchers/4	show	/vouchers/show/4
DELETE	/vouchers/4	destroy	/vouchers/destroy/4
PUT	/vouchers/4	update	/vouchers/update/4
POST	/vouchers/	create	/vouchers/create

Why RESTful?

- Clean URLs. Every URL becomes very easy to understand.
- Format handling. Every resource can easily be requested with different formats.
- Clear code structure. When you open up a controller it is very easy to understand what happens on every request, thanks to the use of the HTTP methods.

Why not RESTful?

- Complications while using AJAX. Sometimes you would like to use the same method for several different outputs. That can however be solved by using formats that is defined by the developer, such as /vouchers.compact to get a compact list of vouchers, however the MIME type is still HTML.
- Sometimes these methods just isn’t enough, you would like to create more actions. You can however do so, but it isn’t totally by the book.