Fsek Documentation

F-sektionen

Oct 17, 2022

Documentation:

1 Learning Web Development 3
- 1.1 How the Web works 3
- 1.2 Learning git 4
- 1.3 Learning resources 4

2 Installing our systems 7
- 2.1 Operating Systems 7
- 2.2 App Installation Guide 8
- 2.3 Web Installation Guide 11
- 2.4 Installing a text editor 17

3 App 19
- 3.1 Standard workflow 19
- 3.2 Our Systems 23
- 3.3 Pages 23

4 Web 25
- 4.1 Useful things to know before using the system 25
- 4.2 File naming and placing 26
- 4.3 Translation files 27

5 Git 31
- 5.1 Git workflow aka git gud 31
- 5.2 Reference 32

6 Code examples 35
- 6.1 Web examples 35
- 6.2 App examples 51

7 Spider Conference 63
- 7.1 Spider Conference 2019 63
- 7.2 Spider Conference 2018 77

8 How to be a spodermon 79
- 8.1 Godkänna användare 79
- 8.2 Create mail alias 79
- 8.3 Ändra mailadresser kopplade till mailalias 80

8.4 Skapa snabblänk . . . 80
8.5 Svara på mail . . . 80
8.6 Gå på möte . . . 80

9 How to prepare the Rostsystem **83**

10 This is a sick title **85**
10.1 This is an ordinary header . . . 85
10.2 Local API . . . 86
10.3 Phonegap . . . 86
10.4 Cloning the Github repository . . . 86
10.5 Creating a custom CMD/Terminal command . . . 87

This is an amazing documentation! Us oldie but goldie spidermen have documented a lot of our work on our various projects to make our code stand the test of time.

CHAPTER 1

Learning Web Development

Getting started with this platform is fairly simple, but there are a few things you need to set up and install. Follow this Installation guide to install the requirements and get set up. We use more systems than mentioned in the Installation guide, but these are already integrated into this repository and do not require any installation. I would recommend reading about all the different systems we use in Our systems later on when you're up and running. It's important to understand these to follow the structure of our project.

If you've installed all the requirements and are new to HTML, CSS and Javascript programming I would recommend going through the information below. Otherwise, visit Standard workflow.

1.1 How the Web works

Read through the following articles and answer the questions associated with each part.

1.1.1 Part 1

- What is a client and a server? What is their relation?
- What is DNS?
- What is DOM?
- Describe in short the process from entering a URL into an address bar to having the webpage loaded

1.1.2 Part 2

Read only "The Client-Server Model" and "A Basic Web App Configuration".

- In what way are HTML, CSS and JavaScript used on a website?

1.1.3 Part 3

Skip “What is REST after all”.

- What HTTP methods are there?

1.2 Learning git

Git is a version control system that makes it possible for a team of developers to work on the same project and edit the same files simultaneously. We do this by having a *master version*, i.e. the current state of the project, online in a *repository* or *repo* on Github’s website. Then, everyone downloads or *clones* the repo and edit their files locally. Note that the changes you make locally will not affect the live master version. Before you make changes though, you need to create a new version of the project or a *branch* that diverges from the master version. That is, we take the project in its current state (master version) and edit it, but we do not save the changes directly to the master branch and update the current state of the project. Rather, we only update our own version on our own branch.

When you’re finished and happy with your changes you want to apply them to the master version and update the project. You do this by first updating our version of the master branch by *pulling* the latest version from the repo. We do this in case someone has updated it while we were making our changes. Then, we apply the master branch on our branch or *rebase* it and see if we have any conflicts with our changes. A conflict appears when the same file is edited in different ways. Github will show us these exact conflicts in the actual files and we can fix them. After that, we upload or *push* our changes to the repo and update our branch online. Now, the last thing we want to do is to apply our changes from our branch to the master branch. We do this by creating a *pull request* that tells the other developers that you want to update the master branch. They can then look at your changes and after they approve, you merge the branches i.e update the master branch with your changes. How all of this is actually done with commands and how you’ll be working with Github is explained in Git workflow.

1.2.1 Codeacademy

- Workflow - https://www.codecademy.com/courses/learn-git/lessons/git-workflow
- Branching - https://www.codecademy.com/courses/learn-git/lessons/git-branching

1.2.2 Questions

- What is a branch?
- Which three commands are used for saving your local changes and putting them on GitHub?
- What is a pull request?
- How does one resolve conflicts with some others changes when rebasing?

1.3 Learning resources

The different languages have very different functions in web development. HTML is the skeleton of the website and the actual building blocks. How these building blocks are positioned, sized and generally styled is what we describe in CSS. CSS could be viewed as the face of the website, it’s what makes it looks good. Lastly, JavaScript is what gives the website life and handles its functionality. It’s with JavaScript that we define the events on button clicks, page loads and more. JavaScript is also where we post and get data from a database. Ruby is the language which Ruby on Rails is built with, which in turn is a web application framework. It contains libraries and tools that allows us to more easily build a website. E.g., does one not have to write the JavaScript code for pressing a button on the website since Ruby on Rails has tools for doing this for us! Below there are some guides and code references to these different languages. I would recommend doing a few guides till you feel comfortable with the structure of a web project and its coding styles. After that, you can take a look at our own project and start familiarizing yourself with that. The best way to get to know our project is by looking at the structure live in a browser while changing things. This is described in step 1-4 in Standard workflow.

1.3.1 Web programming

The following sections contains links to Codeacademy courses and some relevant questions for you to answer for the languages discussed above.

HTML

- https://www.codecademy.com/learn/learn-html/lessions/intro-to-html

CSS

- https://www.codecademy.com/learn/learn-css/

JavaScript

- JavaScript - https://www.codecademy.com/learn/introduction-to-javascript
- jQuery - https://www.codecademy.com/learn/learn-jquery

Questions

- What is the structure in all HTML files and what are their functions?
- What is a `div`?
- What is a `class` and `id` and how can one use them in a CSS file?
- How does an `onClick` event look like in jQuery (JavaScript)

1.3.2 Ruby

- Ruby - https://www.codecademy.com/learn/learn-ruby
- Ruby on Rails - https://www.codecademy.com/learn/learn-rails
- Comprehensive Ruby on Rails book
- Ruby on Rails guides

Read through Understanding the basics of Ruby on Rails HTTP MVC and routes and answer the following questions:

- What does MVC stand for and what are the functions of each part?
- What is the main task of the Rails router?

CHAPTER 2

Installing our systems

Here you can find all information you need regarding the installation of our systems and what operating system to choose when becoming a spoderman.

2.1 Operating Systems

Below are some general information regarding various operating systems and the differences between them. If you wish to learn the reasoning behind our choice of OS, read on, if not continue to *Which OS should I use?*.

Most of your are using either Windows or macOS when starting your career as a spiderman. While these operating systems work fine for every day use, there are better choices when it comes to programming. Many software developers use Linux as their OS of choice due to it's configurability and flexibility. Since it also is, in some sense, a standard in software development, a large amount of tools and frameworks are compatible with Linux.

Linux is a member of the UNIX family which is a collection of operating systems derived from a single old OS from the 1970s. Originally built for programmers the family has expanded to contain many different kinds of operating systems. macOS is also a member of this family and an example of a OS not designed specifically for developers. Due to Linux and macOS both being UNIX operating systems, both work very well when it comes to software development. Windows, however, is built upon MS-DOS, an architecture only used by Microsoft. UNIX and MS-DOS are highly incompatible and as a consequence, UNIX-tools are not guaranteed to work on Windows. Therefore, Windows is generally not a good choice when it comes to software development, unless you are specifically using tools provided by Microsoft or developing software which should work only on Windows (software for Windows can also be developed on Linux or macOS).

2.1.1 Which OS should I use?

- **If you are already using Linux:** Great, you came prepared!
- **If you are using macOS:** This works well too!
- **If you are using Windows:** Won't work, you have a couple of choices, ordered below by level of recommendation.

- *Windows Subsystems for Linux (WSL)*. This method essentially installs Linux on Windows. Weird, yes but it works surprisingly well.
- *Dual booting/changing entirely to Linux*. If you want the most compatability and learn Linux, this is the way to go. It can, however, be a bit of a hassle and take some time. I have currently dual-booted and have no regrets.
- *Using a Virtual Machine (VM)*. If you have a really good laptop with a good graphics card, this can work fine. If not, this method will likely only be a waste of time since the OS will be slow and sluggish.

2.1.2 Windows Subsystems for Linux (WSL)

This might be the easisest and fastest way to get a Linux environment up and running. To install a WSL, simply head to the Windows Store and search for Ubuntu (there are many different Linux versions or distributions but Ubuntu is the most widely used). Simply download and install the app and then you need to run a command in PowerShell. Open PowerShell as an administrator (right click and select *Run as administrator*) and run the following command:

```
Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-Subsystem-Linux
```

Now you should be able to run Ubuntu as a Windows app. If you run into problems, head to this page which contains a troubleshooting section.

The Ubuntu app is simply a terminal which tries to simulate a Linux environment. You can open as many simultaneous instances of the app as you want which can be useful if you want to run several routines at once.

To access files created in the Ubuntu environment, use Visual Code. Download it from https://code.visualstudio.com/, install it and install the extension *Remote - WSL*. You can now open files and folders in Ubuntu with Visual Code.

2.1.3 Dual booting/changing entirely to Linux

Since there are many great tutorials covering this topic, it will not be described in detail here. Just remember to create a backup of your Windows installation before dual booting in case something goes wrong! Also, **Ubuntu** is the most widely used Linux distribution so if you don't know what to choose, that is a good start.

2.1.4 Using a Virtual Machine (VM)

Creating a VM is also covered in great detail in several tutorials online. An open source VM which works okay is VirtualBox. However, there might be some performance issues which could be solved after some tinkering with settings. Another option is VMWare which is not free but there might be some keys available on the web. VMWare has generally worked better for me but in the end, WSL is recommended over a VM since you generally only need a terminal and not an entire GUI.

2.2 App Installation Guide

2.2.1 Installing a UNIX based OS

We recommend using a UNIX based operating system when developing any of our software. If you are using MacOS or a Linux distrubtion you can skip this step and start installing the Adobe PhoneGap CLI. However, if you are Windows user you have some things to do first. No need to worry if you're a Windows lover though, we still love you. There is still time for you to come over and join us in the UNIX world where the grass is greener, especially developing grass. However, it can be a big transition to switch to a completely different OS, so it's understandable if you want to take things slow. Here are you're options:

1. **Uninstall Windows completely and install a Linux distrubtion of your choosing:** I would recommend Ubuntu as a good rebound OS after you dump Windows. Ubuntu is cosy and gives you a nice introduction to Linux. After you're a bit more comfortable you can always switch to something more hardcore. It's completely up to you though.
2. **Dualbooting Windows and Linux:** Allocate space for a Linux parition on your hard drive and have two OS on your machine. This works very well if you want somewhat of a middle ground. It can however be tedious since you have to make sure to not overwrite your original OS.
3. **Install a virtual machine:** Here you will keep Windows completely and rather install a program which runs a virutal Linux desktop enviroment. I recommend trying Oracle's VirtualBox and if that doesn't work well try VMware Workstation Pro.

Assuming you now have a running Unix based OS in some way or another, you can continue on and start installing the actual system below.

2.2.2 Installing the Adobe PhoneGap CLI

The F-app is made with Adobe PhoneGap, so to get up and running we need to install the PhoneGap Command Line Inteface (CLI). We choose not to work with the PhoneGap Desktop App due to its limitations compared to the CLI. The Phonegap CLI has a requirement of both *git* and *node.js*, which we shall install first. Follow the links below and install the software if you haven't done so already.

- node.js - a cross-platform JavaScript run-time environment that executes JavaScript code outside of a browser.
- git - downloads assets in the background for the CLI. Note that some operating systems already have this pre-installed.

Now, we simply install the PhoneGap CLI with Node's default package manager *npm* (Node Package Manager):

```
npm install -g phonegap@latest
```

To check if it has installed properly run `phonegap` in the command line. It should show something similar to:

```
$ phonegap
Usage: phonegap [options] [commands]
Description:
PhoneGap command-line tool.
Commands:
  help [command]       output usage information
  create <path>        create a phonegap project
  ...
```

2.2.3 Setting up Git

We need to configure Git if you have not used it before. Run:

```
git config --global user.name "Firstname Lastname"
git config --global user.email email@example.com
```

using the same email as your account on GitHub.

We also recommended you to run the following command to simplify your pushes to git:

```
git config --global push.default current
```

2.2.4 Cloning the Github repository

If you have never done this before, don't worry, it's fairly simple. You first need to navigate to your desired project location with the `cd` (change directory) command in the command line:

```
cd <preferred project folder>

// Example
cd /home/fredrik/Documents/spindel
```

Then, when use the `clone` command with git, which downloads or *clones* the project from our repository on Github. So, we need to run:

```
git clone https://github.com/fsek/app.git
```

If you've done everything correctly you should see it start downloading files. After it's done you should have a new `app` folder containg all the project files in the directory you chose.

2.2.5 Installing the enviornment

The first thing we need to do after cloning the project is to install the enviornment. First, enter the app directory with:

```
cd app
```

Then run:

```
npm install
```

which will install all required packages for our project. When it's done, execute:

```
npm install gulp-cli -g
```

to get support for *scss*.

Note: On newer versions of macOS (High Sierra or later) you might need to perform some extra steps.

CocoaPods needs to be installed and setup. In `/app` run:

```
sudo gem install cocoapods
pod setup
```

To avoid more manual setup use Node version `11.x`. In the current setup Node versions `12.x`, `13.x`, `14.x` and `15.x` all require different workarounds on macOS.

You will also need to clear and reinstall the phonegap push plugin for ios:

```
rm -r plugins/phonegap-plugin-push
phonegap cordova platform rm ios
phonegap cordova platform add ios
```

2.2.6 Running the server

You have now installed everything and the only thing left to do is to start the server. To do this we simply run:

```
npm start
```

This will start the server and all the required services. The first time you start the web server it will ask you if you want to send information to PhoneGap, which we don't. It will also ask for access through your firewall which you should allow. After a few seconds, you should be able to access the server and see the app at http://localhost:3001. You log in with the email *admin@fsektionen.se* and the password *passpass*

You are now offically up and running. Well done! Head over to *Standard workflow* to get started coding or read more about *Our Systems* to get a better understanding of the project.

2.3 Web Installation Guide

If you are using Windows as your operating system, head over to the *Operating Systems* page before installing the web. Otherwise, assuming you now have a running Unix based OS in some way or another, you can continue on and start following the installation guides below to get set up and running. There is one for Mac users and one for Linux users. Hopefully everything will go smoothly, but sometimes there are some complications since all of our systems are different. They are fixable though, your computer isn't broken. Just ask one of us older spodermen for help and you'll be coding in no time.

2.3.1 Linux installation

The environment requires Ruby 2.5.0, a recent version of Postgres and Redis.

Installing Ruby

Ubuntu

When installing Ruby it's easiest to first install rbenv and ruby-build. Start by installing the required dependencies for your system.

If on Ubuntu run the following commands:

```
sudo apt update
sudo apt install autoconf bison build-essential libssl-dev libyaml-dev libreadline6-
↪dev zlib1g-dev libncurses5-dev libffi-dev libgdbm5 libgdbm-dev
```

On any other Linux distribution just google how to install Ruby and find the requirements there. If you are on ubuntu 20.0.4 or higher, you might need to change libgdbm5 to libgdbm6 in the above command.

Then run:

```
git clone https://github.com/rbenv/rbenv.git ~/.rbenv
git clone https://github.com/rbenv/ruby-build.git ~/.rbenv/plugins/ruby-build
echo 'export PATH="$HOME/.rbenv/bin:$PATH"' >> ~/.bashrc
echo 'export PATH="$HOME/.rbenv/plugins/ruby-build/bin:$PATH"' >> ~/.bashrc
echo 'eval "$(rbenv init -)"' >> ~/.bashrc
exec $SHELL
```

and then install Ruby:

```
rbenv install 2.5.0
rbenv global 2.5.0
```

After this, you should restart your terminal emulator.

Arch

If using Arch, it is easier to use rvm instead of rbenv. rvm requires some packages to work which can be installed by running:

```
 sudo pacman -S base-devel tar gzip diffutils git curl

Download the installation script by running::

 curl -L get.rvm.io > rvm-install
```

then run the script:

```
bash < ./rvm-install
```

and restart your terminal (recommended) or source your .bash_profile by running:

```
source ~/.bash_profile
```

Now confirm that rvm installed successfully by running:

```
type rvm | head -n1
```

which should print:

```
rvm is a function
```

If this is not printed, restart your terminal and try again. If it is still not working, make sure the following line has been added to ~/.bash_profile:

```
[[ -s "$HOME/.rvm/scripts/rvm" ]] && source "$HOME/.rvm/scripts/rvm" # Load RVM into␣
↪a shell session *as a function*
```

Now run:

```
rvm notes
```

too check if the function is working properly. Then run:

```
rvm requirements
```

to install any other dependencies required by rvm. Now run:

```
rvm install 2.5.0
rvm use 2.5.0 --default
```

to install and use the correct ruby version.

Installing Postgres

To install Postgres on a recent version of Ubuntu, it should be enough to run:

sudo apt-get install postgresql postgresql-contrib libpq-dev

If you are using the LTS (Long Term Support) version, you need to run the following commands instead:

```
sudo sh -c "echo 'deb http://apt.postgresql.org/pub/repos/apt/ xenial-pgdg main' > /
↪etc/apt/sources.list.d/pgdg.list"
wget --quiet -O - http://apt.postgresql.org/pub/repos/apt/ACCC4CF8.asc | sudo apt-key␣
↪add -
sudo apt-get update
sudo apt-get install postgresql-common
sudo apt-get install postgresql-9.5 libpq-dev
```

On most other distributions, it's enough to install the postgresql or postgres package. For example, on Arch:

```
sudo pacman -S postgresql
```

Now, to start postgres on Ubuntu computers:

```
sudo systemctl start postgresql
```

If you are using WSL (version 1):

```
sudo service postgresql start
```

This is because WSL version 1 does not use systemd and can therefore not use systemctl.

We probably want to make sure postgres starts on startup, like this:

```
sudo systemctl enable postgresql
```

I have not found a nice way to get postgresql to autostart on WSL.

To use Postgres with Rails you need to create a user:

```
sudo -u postgres createuser <username> -sP
```

Postgres will then ask you to set a password for the new user.

Installing Redis

Redis can usually be installed with your distribution´s package manager. It's often called either redis-server or just redis. On Ubuntu just run this command:

```
sudo apt-get install redis-server
```

On Arch, you just run:

```
sudo pacman -S redis
```

On Ubuntu we also want to stop a current running redis server, since we want to use it for ourselves. We do this by running:

```
sudo systemctl stop redis-server
```

If you are using WSL (version 1):

```
sudo service redis-server stop
```

Redis starts by itself on startup, so we need to stop it like above every we want to use it. To prevent it from starting by itself and make our lives easier, we simply run:

```
sudo systemctl disable redis-server
```

This might not be specific for Ubuntu, so if the server doesn't start with `foreman s` later on, come back here and disable redis. That might fix it.

Setting up Git

You need to configure Git if you have not used it before. Run:

```
git config --global user.name "Firstname Lastname"
git config --global user.email email@example.com
```

using the same email as on GitHub.

You are recommended to run the following command to simplify pushes to git:

```
git config --global push.default current
```

Installing the environment

To install the environment you should first clone the repo. Head to your preferred directory and clone. Afterwards you need to install Rails and all the gems required. All these things can be achieved by running the following commands:

```
cd <preferred folder>
git clone https://github.com/fsek/web.git
cd web

gem install bundle
bundle install
```

If bundle install throws an error then run the follwing command first (observed on WSL version 1):

```
gem update --system
bundle install
```

To run Rails and store data you need to configure the database connection. In the environment root folder there is a file called .env-sample. Copy this file and rename it to .env:

```
cp .env-sample .env
```

Now open the `.env` file in your favourite text editor and enter the username and password you chose when creating a Postgres user. Enter the same username and password for both the test and dev environment.

Make sure the `sidekiq.log` file exists in the *web/log* directory. You can do this by running:

```
ls ./log
```

and then see if `sidekiq.log` shows up. If not we need to create it with:

```
touch log/sidekiq.log
```

otherwise you can continue on.

We also need to generate a "Secret key base" for Rails. Run:

```
echo "SECRET_KEY_BASE=$(rails secret)" >> .env
```

You are now ready to load the database structure into Postgres, and populate it with some example data. Run the following commands:

```
rails db:create && rails db:migrate && rails db:seed && rails db:populate_test
```

Additional packages

For image upload to work properly on your local machine you need Imagemagick:

sudo apt-get install imagemagick

Running the server

To run the server and all the required services simply run the command:

```
foreman s
```

After a few seconds, you should be able to access the server at http://localhost:3000. You log in with the email *admin@fsektionen.se* and the password *passpass*.

2.3.2 Mac installation

The environment requires Ruby 2.5.0, a recent version of Postgres and Redis.

Installing Ruby

When installing Ruby it's easiest to first install rbenv and ruby-build. Start by checking that you have the required dependencies for your system.

Then run:

```
git clone https://github.com/rbenv/rbenv.git ~/.rbenv
git clone https://github.com/rbenv/ruby-build.git ~/.rbenv/plugins/ruby-build
echo 'export PATH="$HOME/.rbenv/bin:$PATH"' >> ~/.bash_profile
echo 'export PATH="$HOME/.rbenv/plugins/ruby-build/bin:$PATH"' >> ~/.bash_profile
echo 'eval "$(rbenv init -)"' >> ~/.bash_profile
exec $SHELL
```

and then install Ruby:

```
rbenv install 2.5.0
rbenv global 2.5.0
```

After this, you should restart your terminal emulator.

Installing Postgres

To install Postgres we use Homebrew and it should be enough to run:

```
brew install postgresql
```

Now, to start postgres:

```
brew services start postgresql
```

To use Postgres with Rails you need to create a user:

```
createuser <username> -sP
```

Postgres will then ask you to set a password for the new user.

Installing Redis

Redis can usually be installed with your distribution´s package manager. It's often called either redis-server or just redis. Just run this command:

```
brew install redis-server
```

Setting up Git

You need to configure Git if you have not used it before. Run:

```
git config --global user.name "Firstname Lastname"
git config --global user.email email@example.com
```

using the same email as on GitHub.

You are recommended to run the following command to simplify pushes to git:

```
git config --global push.default current
```

Installing the environment

To install the environment you should first clone the repo. Head to your preferred directory and clone. Afterwards you need to install Rails and all the gems required. All these things can be achieved by running the following commands:

```
cd <preferred folder>
git clone https://github.com/fsek/web.git
cd web

gem install bundle
bundle install
```

To run Rails and store data you need to configure the database connection. In the environment root folder there is a file called .env-sample. Copy this file and rename it to .env:

```
cp .env-sample .env
```

Now open the `.env` file in your favourite text editor and enter the username and password you chose when creating a Postgres user. Enter the same username and password for both the test and dev environment.

Before you can continue, Rails wants you to generate a “Secret key base”. Run:

```
echo "SECRET_KEY_BASE=$(rails secret)" >> .env
```

You are now ready to load the database structure into Postgres, and populate it with some example data. Run the following commands:

```
rails db:create && rails db:migrate && rails db:seed && rails db:populate_test
```

Running the server

To run the server and all the required services simply run the command:

```
foreman s
```

After a few seconds, you should be able to access the server at http://localhost:3000 You log in with the email *admin@fsektionen.se* and the password *passpass*.

2.4 Installing a text editor

To actually start coding you also need a text editor, much like Eclipse for Java. For this project, I would recommend using Sublime Text. Sublime is a solid editor that is very straightforward, easy to set up and is what I use personally. If you have another editor that you prefer, that's perfectly fine too. Whatever editor you end up using just make sure to use our standard and set the tab width to two and indent using spaces (soft tabs). In Sublime you do this by going to Preferences > Settings, which should open a new sublime window with two columns. In the left column, there are the default settings and in the right your own personal settings. In the personal column, you want to add:

```
"tab_size": 2,
"translate_tabs_to_spaces": true,
"ensure_newline_at_eof_on_save": true
```

Note, that you need to put a comma at the end of each line (except the last one) for the settings to apply.

You have now installed everything that you need to get started. However, before you can continue you also need to be able to write some actual code. The F-app is built with HTML, CSS, and Javascript and if you're not familiar with these languages I would recommend going through some guides/tutorials here. Otherwise, we'll take a look at the Standard workflow.

CHAPTER 3

App

Welcome to the F-guild's app or the infamous F-app!

We're very glad that you want to work on our platform, but before you can start there are a few things you need to do. If you are new to web development I would highly recommend to visit *Learning Web Development* for various tutorials and general learning material. It's good to have a basic understading of the web and web development before getting into a more complex project, like the F-app. Otherwise, go ahead and follow the *Installation guide* to get setup and running.

If you're already set up visit Standard workflow to learn more about how you actually work.

If you wish to learn more about the different systems we use you can read about them in the Our systems page.

We wish you the best of luck in your programming ventures. We're certain that you'll do great! Remember that you can always ask any of us older spodermons for help if you ever run into trouble. We normally don't bite, so don't be afraid.

3.1 Standard workflow

If you've followed the *App Installation Guide* you should have everything set up to start working. Below we're going to take a look at the standard workflow on this project. This will help you get started coding and implementing your feature.

3.1.1 Opening the project folder

This step might seem a little trivial, but it's important to make sure that we have all the correct files opened going forward. Go ahead and open the `app` directory in your IDE/text editor of choice. The folder that's important is the `www` folder, which contains all of our HTML, SCSS/CSS and Javascript files. This is where you'll be doing all of your work and adding your feature. The folder structure in `www` is sorted according to the different file types and is quite self explanatory.

3.1.2 Creating a new branch

With the project opened you are ready to start coding. Before we make any changes though, we want to create and checkout a new branch with git.

```
git checkout -b <branch name>
```

If this is new to you I would recommend to checkout *Learning git* before we continue and familiarize yourself with the git workflow and its core concepts.

3.1.3 Starting the web server and running the app

Now we are good to start coding and contributing to the project. Remember that all the changes you make are local and doesn't affect anyone but you. Now, it would also be quite nice to be able to see the effect of the changes you make. We do this by hosting a local web server with PhoneGap that runs the app for you in real time. To start the server we use npm and simply execute:

```
npm start
```

The first time you start the web server it will ask you if you want to send information to PhoneGap, which we don't. It will also ask for access through your firewall which you should allow. After a few seconds, you should be able to access the server and see the app at http://localhost:3000. Make sure you open the developer tools in your browser (F12), change to a responsive design mode (Ctrl+Shift+M) and select a device.

3.1.4 Making changes

This is where we part for a while and where you'll shine bright like the star you are. Go ahead and start coding that sik feature with those golden fingers. I recommend reading through *Our Systems* before you start. This will give you a better understanding of the project and help you get the ball rolling. Below are. . . . Assume that you have a basic understanding of Framework7.

Creating a new page

The first thing we need to is add a new HTML file that will contain our new page. We can do this through a text editor/IDE or with the command line. Do what ever works for you, but it's important to have the `.html` file ending and make sure to save it directly in the `www` folder, with all the other HTML files. Lets assume you have created a page called `cool_stuff.html`. Now, we obviously want to add some cool stuff, but lets start with the standard page layout:

```
# app/www/cool_stuff.html
<div data-name="cool-stuff" class="page no-toolbar">
  <div class="navbar">
    <div class="navbar-inner sliding">
      <div class="left">
        <a href="#" class="back link">
          <i class="icon icon-back"></i>
          <span class="ios-only">Tillbaka</span>
        </a>
      </div>
    </div>
  </div>
  <div class="page-content">
    Some cool content...
```

(continues on next page)

(continued from previous page)

```
  </div>
</div>
```

This is more or less the structure all of our pages follow, with the important difference of `data-name`. The `data-name` is very important because it gives your page a unique tag that we use to catch different page specific events. , like *onPageInit* or ‘ (F7-link).

Okay, so we have created a pretty cool page, or atleast a foundation for one, but we have no way of navigating to it yet. This is fairly simple to fix and consists of two parts: *defining a new route* and *linking to a route*. To define a new route we have to open `index.js` and scroll down to the different view objects. Views can be seen as HTML objects that contain a selection of pages and a page navigation history. The app contains six views, one for each tab and one for the login page. You can read more about views **Our systems link and F7 link???**. Let's say we want to add our cool page to the alternatives tab. This means that we need to add a new route to `cool_stuff.html` in the `alternativesView` object.

```
# app/www/js/index.js
var alternativesView = app.views.create('#view-alternatives', {
  routesAdd: [
    {
      name: 'cool_stuff',
      path: '/cool_stuff/',
      url: './cool_stuff.html',
    },
    {
      ... # Other routes
    },
  ]
});
```

Now the F7 router knows which page to load when we navigate to the `/cool_stuff/` route in the alternatives view. We can now begin on the second part, linking to a route. That is, we can setup a HTML link to our page and the F7 router will lookup the correct file to render through the route we just defined. All we need to do is define the route in the `href` attribute of the `<a>`-tag:

```
<a href="/cool_stuff/">Link to a cool page</a>
```

(Read more about routes blabla, simplest form - from root, second link needs to defined in the routes of the route.) Let's add a link to the alternatives tab in `index.html` under “Sångbok”. Here we add a bit more to the link to make it a list item, rather than just text.

```
# app/www/index.html
<div id="view-alternatives" class="view tab">
  <div data-name="alternatives" class="page">
    <div class="navbar android-hide">
      <div class="navbar-inner sliding">
        <div class="title">Alternativ</div>
      </div>
    </div>
    <div class="page-content settings-content">
      <div class="list">
        <ul>
          <li>
            <a href="/songbook/" class="item-link">
              <div class="item-content">
                <div class="item-inner">
                  <div class="item-title">Sångbok</div>
```

(continues on next page)

(continued from previous page)

```
                </div>
              </div>
            </a>
          </li>

          # Link to cool_stuff.html
          <li>
            <a href="/cool_stuff/" class="item-link">
              <div class="item-content">
                <div class="item-inner">
                  <div class="item-title">A cool page</div>
                </div>
              </div>
            </a>
          </li>

          ... # More list items and stuff

        </ul>
      </div>
    </div>
  </div>
</div>
```

That's it. It should now be possible able to navigate to `cool_stuff.html` from the alternatives tab. Now we want to fill the page with some cool stuff, which often is done with content retrived through our API.

Making an API request

Intro om intressant innehåller => API + databas

Let's continue with `cool_stuff.html` from the example above and try to add some cool stuff with the API. Before we get deeper into the actual request, we first need to figure out when we want to make this request and where it should be defined and executed in our code. The first question has a pretty If we assume that there exists an endpoint in our API called `cool_stuff` which has a GET-request set up. Now,

Creating a template

3.1.5 Testing on multiple devices and operating systems

When you have gotten to the styling part of your feature make sure you test on both Android and iOS devices, since they have different styling. Also, remember to switch the OS specific styling overrides in `index.html` when you switch OS. That is, make sure you have the `material-overrides.css` when working with Android devices and `ios-overrides.css` on iOS devices. If you don't, you'll notice that things look very bad. It's also important to test on different screen sizes. What looks good on one device doesn't necessarily look good on others.

3.1.6 Adding changes to next version/update

When you're finished with your changes and they look good on all devices it's time to add them to the next version. We do this using git which is described step by step in *Git workflow aka git gud*. Head over there and follow the steps. When your done the whole section will enjoy your feature in the next release.

3.2 Our Systems

TODO: Fixa denna sidan och app.rst sen PR po de.

3.2.1 PhoneGap

Phonegap is kewl

3.2.2 Framework 7

Framework7 is a framework for building iOS and Android apps in HTML. This is already installed and integrated into this GitHub repository, so you don't need to install anything locally to start using it. However, it is still a good idea to look through Framework7's docs to familiarize yourself with their different components and our app's structure. Almost all of our components in the app comes from Framework7, so if you ever wonder what certain things are or how they work you can most likely find information in their docs.

3.2.3 Template 7

Template 7

3.2.4 J-toker

J-toker

3.3 Pages

All the pages in the app.

3.3.1 Songbook

This is the **best** and the most *important* page of the app.

Javascript Description

songbook.js contains the JS code for both the songbook page **songbook.html** and for the individual song pages song.html.

JS for the songbook

Upon initializing the songbook page the API-endpoint **/songs** (link to the API description table) is accessed and the JSON object is retrieved. The JSON contains the songs in alphabetical order with the song id, text, author and category. When the data is retrieved the function **initSongList** is called.

The **initSongList** function regroups the songs according to the first letter and builds the html from the template of the songbook. The first letter of the first song is stored in the first attribute *firstLetter* of a JSON object. Then all the songs with the same first letter are stored in an array which is then stored as the second attribute of the JSON object. For every first letter, a JSON object is created containing the actual letter and an array of the songs. These JSON objects are then pushed into the array **songList**. The songList object is structured in the following manner:

```
[{firstLetter: A, songs: [song1, song2, ...]}, {firstLetter: B, ..}, ...]
```

initSongList then calls on the template **songbookTemplate** with the **songList** as parameter an the template creates the html for the songbook page. The letter index is then created using the Framework7 object **listIndex**:

```
var listIndex = app.listIndex.create({
  el: '.list-index',
  listEl: '#songbook-list',
  label: true,
});
```

JS for the individual songs

The song objects in the songbook list links to **song/id**. When the song page is initialized, the API-endpoint **/songs/id** is called. This returns a JSON with the song text and some additional info about the song. The template is then created from the JSON information and added to the song page html.

CHAPTER 4

Web

Web stuff

4.1 Useful things to know before using the system

The website is programmed with a number of different languages with different functionalities. Depending on what you want to do you will have to learn some of the languages. The main distinction between the languages is if they run on the server or the client. If you want to do design and user interaction you will want to learn the client-side languages and if you prefer doing backend functionality you want to learn the server-side languages. Mostly when working on the website you will do both frontend and backend since most projects involve the client and server working together.

4.1.1 Server-side

When programming the server-side you will mostly use Ruby with the Rails ecosystem. It is therefore recommended to learn a bit of Ruby before attempting backend project. You will also have to get used to the Rails ecosystem.

Recommended learning resources and code reference

(länkar)

Client-side

The client-side is mainly about user interaction. The languages you will use client-side are HTML, CSS and Javascript. HTML is usually not called a programming language but rather a markup language. It is used when structuring the visible content on the website. To style the content we use CSS and to add interactivity like drop-down menus we use Javascript.

Every language uses frameworks to ease the development process. Our HTML-files are prefixed with .erb to allow usage of Ruby code in the files. With CSS we use SCSS to allow variables and imports. Javascript uses jQuery to simplify programming user interaction.

Recommended learning resources and code reference

(länkar)

4.2 File naming and placing

Hopefully, you have learned a bit about Rails and how things work together. If not head over here to learn a bit. This guide will make more sense if you know a bit about Rails.

4.2.1 Help Rails

Rails take care of a lot of things for you. But to help Rails you need to work in a specific way. Rails look for certain things while compiling all the files you've written. To help Rails you need to place your files in specific folders and name them in specific ways. If everything is done correctly Rails will also compile your files correctly.

4.2.2 File Placement

All files are placed in subfolders located in app. Generally, images and other assets are placed in the assets folder.

Rails

- Models
  - Models are placed in the models folder.
- Controllers
  - All controllers are placed in the controllers folder. If you want to restrict usage to users with higher privileges you place the files in controllers/admin.
- Views
  - A view is a folder of files which is placed in the views folder. Same as with the controllers, if you want to restrict access to regular users then place them in the admin subfolder.

CSS

All style files should be placed in the assets/stylesheets/partials folder.

Javascript

Javascript files are placed in the assets/javascripts folder.

4.2.3 Naming

To make this section more understandable a bike management system will be pseudo-implemented.

Rails

- Models
  - Name your model something descriptive. You will use the model name later when naming the controller and view. Let's create a model named bike.rb.
- Controllers
  - Name your controllers with [model-name]_controller but the model name in the plural form (it's not really plural in all cases as you just add an s to the model name). In our case, the controller will be called bikes_controller.rb.
- Views
  - Use the model name in the plural form when naming your view. In our case, our view folder will be called bikes.

CSS

The most important rule when naming CSS-files is to use an underscore (_) before the name. Other than that you will want to name the file something descriptive. Rails don't really care what you name your file but other developers will. In our case, the file would, for example, be called _bikes.scss

Javascript

Give the JS-files descriptive names. Rails don't care about the name but others will. In our case, the name would be bikes.js.

4.3 Translation files

We are using the platform OneSky to handle our Swedish and English translations. This section describes how to use the gem onesky-rails we are using to talk to the OneSky API as well as the workflow you should follow when adding a new phrase.

4.3.1 Adding OneSky API secret to environment

The gem 'onesky-rails' provide a `upload` and `download` rake command to sync files. To be able to use these you must add the public and secret key from our OneSky account locally to your computer, more specifically to the environment file `.env` inside the `web` folder. These can be found by logging onto the OneSky website. Ask your foreman for login details. Once you have logged in press "Site settings" on the upper right and then "API Keys & Usage" to find the keys.

Before adding the keys to your computer you must first change your current directory to the `web` folder. When you are there run

```
echo ONESKY_API_KEY=PUBLIC_KEY >> .env
echo ONESKY_API_SECRET=SECRET_KEY >> .env
echo ONESKY_PROJECT_ID=67804 >> .env
```

with the keys from the OneSky website to add the keys to the `.env` file.

4.3.2 Workflow

This section describes the standard workflow of adding a new phrase.

Add phrase to Swedish translation file

The translation files can be found in `web/config/locales`. Model specific phrases and phrases used in the views can be found inside the folder `models` and `views` respectively.

Example

Here follows a part of the translation file `meetings.sv.yml` used in the meeting view:

```
sv:
  meetings:
    index:
      new: Ny bokning
      to_alumni: Till alumnirummet
      to_sk: Till styrelserummet
      administrate: Administrera
    edit:
      title: Redigera bokning
      destroy: Förinta bokning
      index: Alla lokalbokningar
      confirm_destroy: Är du säker på att du vill förinta bokningen?
```

Each phrase has a so called Phrase ID. The ID of the last row above is `meetings.edit.confirm_destroy`. It is important that this ID is named properly for the meeting view to be able to find the phrase. The last attribute, e.g. `confirm_destroy`, should say something about what the phrase is about or where it is used.

Say you want to add a phrase to the edit page of meetings. To do this you simply add a line with a suitable attribute as

```
sv:
  meetings:
    index:
      new: Ny bokning
      to_alumni: Till alumnirummet
      to_sk: Till styrelserummet
      administrate: Administrera
    edit:
      title: Redigera bokning
      destroy: Förinta bokning
      index: Alla lokalbokningar
      confirm_destroy: Är du säker på att du vill förinta bokningen?
      new_phrase: Ny fras
```

Upload translation files to OneSky

When you have added the phrase to an already existing or a new translation file, we want to upload them to OneSky. This can be done by running

```
rake onesky:upload
```

inside your `web` folder. Note that this command will upload all Swedish translation files, including those that have not been edited.

Add English translation to phrase

Now we need to add an English translation to the added phrase. This is done by logging onto the OneSky website and going to “Projects->Fsektionen/Regular Web App” and then selecting “English”. From this page you can filter to only show the relevant translation file, e.g. `meetings.sv.yml`. OneSky should then show you which phrases that are not yet translated.

Export English translation

Now press the F-guild logo to the upper left to goto to the home page and then press “Download translations”. Choose `.yml (Rails YAML)` as the file format and filter to only export the English translation of the relevant file.

Once the export is done, simply open the downloaded translation file with your favourite text editor and copy the content of the file and paste it into the corresponding English translation file in `web/config/locales/onesky_en`.

CHAPTER 5

Git

This where you git gud.

5.1 Git workflow aka git gud

The steps below describe going from an idea, implementing it and then adding it to the live website. If you're new to Github I would recommend heading here for a more general description of the workflow. It's important to understand the purpose of the steps below before you learn the actual commands.

5.1.1 1. Create a new branch: git checkout -b new-branch

This step ensures your changes won't interfere with the main branch of the website. When on your own branch you can change stuff without worrying about accidentally affecting the website.

5.1.2 2. Make some changes

5.1.3 3. Prepare to upload changes: git add [FILENAME] [-FLAG]

This command tells git that you are finished editing your files. The FLAG is optional and is useful if you want to add all changed files. If that is the case omit FILENAME and use A as flag.

5.1.4 4. Group changes and describe them: git commit [-m "message"]

Try to write a descriptive message to ensure changes make sense to any collaborator. If you write only git commit git will use a text editor where you can write your commit message. This allows multiline messages which can be useful.

5.1.5 5. Head to the master branch to ensure you have the latest version of the website: git checkout master

It's important to get the latest version to check if your changes interfere with any other changes.

5.1.6 6. Fetch changes: git pull

Downloads the latest version from GitHub. If no changes were downloaded go to step 11.

5.1.7 7. Return to your branch: git checkout [YOUR BRANCH]

5.1.8 8. Combine the changes to master with yours: git rebase master

Before adding the changes to the live website it's important to combine them with other changes. Doing this ensures no complication arises when later adding your changes. You will most likely get an "error" message saying your changes could not automatically be merged. If that is not the case then go to step 11.

5.1.9 9. Solve conflicts by doing any of the steps below

Edit each conflicting file and choose the changes you want to keep You can choose an entire file from either master or your branch by running: git checkout [OURS/THEIRS] - [FILE NAME] Note: THIS IS HIGHLY UNINTUITIVE OURS: This will use the file from master THEIRS: This will keep the file from your branch Yes this is unintuitive but ours and theirs are from the master branch point of view.

5.1.10 10. When you feel that you have solved all conflicts run (if this fails, keep fixing conflicts with step 9): git rebase –continue

5.1.11 11. Push changes to GitHub: git push (-fu)

If you get an error about being behind the remote branch and still know you want to push run the command with the flag -fu.

5.1.12 12. Head over to GitHub and change to your branch. Create a pull request and describe your changes to potential reviewers. Make changes after suggestions from collaborators and bots.

5.1.13 13. If you have several commits which can be logically grouped. Squash them with git rebase -i HEAD~x, where x is the number of commits you wish to change. You are not required to change x commits, you simply have the possibility.

5.1.14 14. When you and your collaborators feel satisfied with the changes rebase them into master.

5.2 Reference

Alla kommandon prefixas med "git"

5.2.1 Allmänna kommandon

- **Starta rep:** init
- **Klona rep:** clone [url]
- **Fixa remote branch:** config –global push.default current
- **Se alla commits i lista:** log
- **Visa ändringslogg (hitta commit-id ifall du tappar bort något):** reflog

5.2.2 Skicka och hämta filer

- **Skicka ändringar till remote:** push
- **Skicka första gången:** push -u
- **Tvinga skickandet:** push -fu
- **Hämta ner ändringar från servern:** pull
- **Hämta ner ändringar men slå inte ihop med lokala filer:** fetch

5.2.3 Filhantering

- **”Spara” filer i git:** add
- **”Spara” alla ändrade filer:** add -A
- **Visa filer som ändrats:** status
- **Ta bort fil ur git:** git rm [filnamn]
- **För in fil från annan branch:** checkout [branchnamn] – [filnamn]
- **Klumpa ihop alla filer du lagt till med ”add” för att skicka till remote:** commit (-m [meddelande])
- **Lägg till filer i den förra klumpen:** commit –amend
- **Titta på tidigare commits och sammanfoga commits etc.:** rebase -i HEAD~5
- Byt från pick till andra meddelanden för att byta namn eller sammanfoga commits.

pick: Låt committen vara kvar **fixup:** Slår ihop med närmaste pick uppåt

5.2.4 Branches

- **Gör en ny branch (en ny version av hemsidan som du kan arbeta i):** branch [branch-name]
- **Visa alla lokala branches:** branch
- **Ta bort en branch lokalt och remote:** branch -D
- **Ta bort en branch lokalt:** branch -d
- **Byt branch:** checkout
- **Ny branch och byt:** checkout -b [namn]
- **Sammanfoga branches på dåligt sätt:** merge [annan branch]

- **Sammanfoga branches på bra sätt:** rebase [annan branch]
- **Återställ branch till annan branch:** reset –hard [commit eller branch]
- **Återställ men ändringar läggs till med add i stage:** reset –soft [commit eller branch]
- **Återställ men ändringar kan läggas till med add om man vill:** reset –mixed [commit eller branch]

5.2.5 Workflow

Du vill skriva lite kod och skicka upp till servern

1. **Gör ny branch:** git checkout -b ny-branch
2. **Skriv din kod**
3. **Lägg till ändringar i stage:** git add -A
4. **Klumpa ihop ändringar och beskriv dem:** git commit
5. **Skriv ett meddelande som beskriver ändringarna**
6. **Gå till master-branchen för att se om någon annan ändrat något:** git checkout master
7. **Hämta ner eventuella ändringar:** git pull
8. **Gå till din egna branch för att sammanfoga ändringarna:** git checkout ny-branch
9. **Hämta in ändringar från master:** git rebase master

#. **Lös konflikter genom att gå igenom manuellt eller kör kommandot:** git checkout XXX – [filnamn] i. XXX = ours för att ta filen från den andra branchen ii. XXX = theirs för att ha kvar filen #. **Fortsätt med sammanfogningen (om det inte går fortsätt med steg 10):** git rebase –continue #. **Skicka filer till branchen på remote:** git push (-fu om det inte fungerar) #. **Gå in på github och skapa en pull request där du beskriver vad du gjort och gör eventuella ändringar som bottar och personer föreslår.**

CHAPTER 6

Code examples

Here you can find some walkthroughs of code examples. These give knowledge on the basic workflow and also provide some understanding of underlying code mechanics.

6.1 Web examples

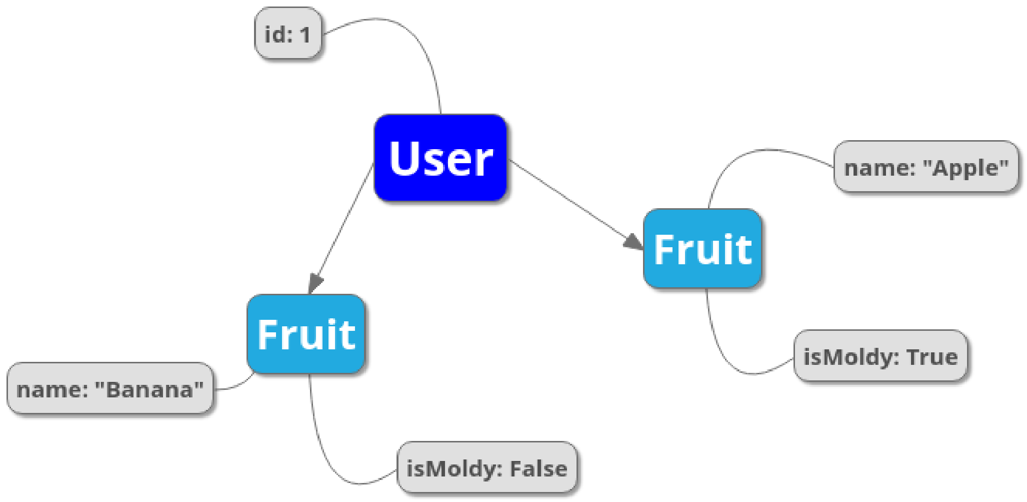
Here are some examples of web code.

6.1.1 Fruits

This is a highly explanatory and beginner-friendly example of web implementation. Even though it's a little silly, it explains the workflow and basic process of implementation quite well.

General idea

We begin by creating an overview of the idea. For this example, the vision is to implement the possibility of allocating fruit to the users on the website. Before we begin writing any code we must think about the general properties of a fruit, given the context.



So, generally we want to create a fruit object described by the two parameters name, a string, and is_moldy, a boolean. The fruit is connected to a user of whom it's owned by. A user can own many fruits, but a fruit cannot be shared between users. These kinds of relations are important to keep track of.

Create a git branch

Before actually writing any code, you must create a git branch so as to not accidentally mess with any other code (and especially not with the master branch). It is praxis to name it using kebab case in a manner of **[your name]-[general description]**. We create a new git branch through the command

```
git checkout -b "johanna-fruits-example"
```

you can then make sure you are on the right branch through the command

```
git status
```

or

```
git branch
```

Migrate the database

For each Rails model, there is a database table. We go to **web/db/** and create a new file called **20201129220000_create_fruits.rb** as we are about to create a database for the fruits on 29 November 2020 at 10 pm. This naming convention exists to structure the files according to their creation.

Here we will specify what parameters there are.

```
class CreateFruits < ActiveRecord::Migration[5.0]
  def change
    create_table :fruits do |t|
      t.references :user
      t.string :name, null: false
      t.boolean :is_moldy, null: false, default: false
    end
  end
end
```

We are saying that each fruit inhabits a name and degree of moldiness, neither of which can be empty (`null`), and is connected to a certain user. We have set the default value of `is_moldy` to false, as we would expect a new fruit to be fresh.

When this is finished we go to the terminal and run

```
rails db:migrate
```

Create a model

This time the file and class name are singular, as opposed to the database elements that are plural. We go to **web/app/models/** and create a new file called **fruit.rb**.

```
class Fruit < ApplicationRecord
  belongs_to :user, required: true
  validates :name, presence: true
  validates_inclusion_of :is_moldy, in: [true, false]

  def to_s
    name
  end
end
```

Here we once again see the different parameters. In the model we make sure that the right values go into the fruits' database table.

First we specify the relation between a user and a fruit; since we wish the user to own fruits, and a fruit to be owned by a single user, we use the line `belongs_to` which describes it quite well. `belongs_to` is an Association that makes the creation and deletion of objects smoother.

The `validates :name, presence true` line ensures that a fruit only can be created if it is given a name. Same goes for `is_moldy`. But how come we don't write `validates :is_moldy, presence: true`? Doing so will lead to some bugs when creating a fruit. Read the documentation for `validates`:

If you want to validate the presence of a boolean field (where the real values are true and false), you will want to use validates_inclusion_of :field_name, in: [true, false].

This is due to the way Object#blank? handles boolean values: false.blank? # => true.

Lastly there is the `to_s` function, which is quite self explanatory.

Test the model directly through the Rails console

At this point the fruit is practically done, console-wise. It is very practical to continuously try out an object directly through the Rails console while it is being implemented. Run

```
rails c
```

to enter the Rails console. We copy the situation in the top illustration by running

```
Fruit.create!(user_id: 1, name: "Banana", is_moldy: false)
```

and

```
Fruit.create!(user_id: 1, name: "Apple", is_moldy: true)
```

The user with user_id: 1 (Hilbert Admin-älg) now owns two fruits. You can run

```
Fruit.all
```

to ensure that it is a list containing two fruits with the correct parameters. To delete these fruits we run

```
Fruit.delete_all
```

Add an association to user

We have already declared the Association `belongs_to` for the fruit, but we also need to declare a related Association for the user. We go to **web/app/models/user.rb** and write the following line

```
has_many :fruits, dependant: :destroy
```

which, of course, says that a user can own many fruits. The `dependant: :destroy` bit is what ensures that all associated fruits will vanish as the user is deleted. If we go back to the Rails console, we can try out some new things. This time we will create the same fruits, but instead of having the user_id as a parameter, we will create the fruits directly through the user

```
User.first.fruits.create!(name: "Banana", is_moldy: false)
User.first.fruits.create!(name: "Apple", is_moldy: true)
```

Then calling

```
User.first.fruits
```

will return a list of these two fruits. We write `User.first` since we want to reach the first element in the list of users. Writing `User.find(1)` returns the user with `id` 1, and is equivalent to `User.first`.

Define the routes

In order for the fruits to show up on the website, the different routes have to be initialized in the file **web/app/config/routes.rb**. Before adding any code we have to be sure about who is supposed to have access to what. For this example we would like each user to be able to view their own fruits, and only admins to be able to create and delete fruits. We will therefore write

```
resources :fruits, path: :frukter, only: [:index, :show]

namespace :admin do
  resources :fruits, path: :frukter, except: :show
end
```

and under the api namespace (`namespace :api`), add:

```
resources :fruits, path: :frukter, only: [:index, :show]
```

We can view all the available fruit-paths by running

```
rails routes | grep fruit
```

If we were to run rails routes only we would get an endless stream of every single route.

Create the controllers

The admin controller

As we have specified that there are going to be different routes for admins and regular users, there has to be different controllers for each. We begin by writing the admin controller. We go to **web/app/controllers/admin/** and create a file **fruits_controller.rb**. The convention here is to name the file in plural. The "shell" of the file looks like this:

```
class Admin::FruitsController < Admin::BaseController
  load_permissions_and_authorize_resource

  [methods]

end
```

The class is also named in plural, and doing so has the benefit of automating some default routes. As you might have figured out, the **FruitsController** inherits from the **Admin BaseController**. You can view its contents at **web/app/controllers/admin/base_controller.rb**, if you're curious.

In the controller we write methods that will be used to execute actions concerning the fruit. What actions do we want admins to be able to perform?

- Retrieve all existing fruits
- Create a new fruit
- Edit a fruit
- Delete a fruit

The control methods are quite standard, so let's take a look at the finished file and then analyze its contents.

```
class Admin::FruitsController < Admin::BaseController
  load_permissions_and_authorize_resource

  def index
    @fruits = initialize_grid(Fruit)
  end

  def create
    @fruit = Fruit.new(fruit_params)
    if @fruit.save
      redirect_to(admin_fruits_path, notice: alert_create(Fruit))
    else
      render :new, status: 422
    end
  end

  def update
```

(continues on next page)

(continued from previous page)

```
    if @fruit.update(fruit_params)
      redirect_to(edit_admin_fruit_path(@fruit), notice: alert_update(Fruit))
    else
      render :edit, status: 422
    end
  end

  def destroy
    @fruit.destroy!

    redirect_to(admin_fruits_path, notice: alert_destroy(Fruit))
  end

  private

  def fruit_params
    params.require(:fruit).permit(:name, :is_moldy, :user_id)
  end

end
```

Let's go back to the bullet list from before and match it with the corresponding methods:

- Retrieve all existing fruits - *index*
- Create a new fruit - *new and create (and fruit_params)*
- Edit a fruit - *edit and update*
- Delete a fruit - *delete*

Standard control methods

But wait, how can there be new and edit methods if we haven't implemented them ourselves? The way Rails works enables us to leave methods empty if we don't want it to do something special. Rails, by standard, renders the corresponding view (we'll get to that later) when one navigates to a url; so clicking on an "edit" button will in this case render **web/app/views/edit.html.erb** and nothing else.

The @fruits variable

`index` initialize the variable `@fruits`. As for `edit`, `index` will render its corresponding view, but with the difference that we have made `@fruits` accessible in there. `@fruits` is given the value `initialize_grid(Fruit)` since we wish to render a grid of the Fruit database table in the index view. For `update` and `delete`, it is possible to use the `@fruit` variable without initializing it as the preceding action (e.g. pressing a button) itself will make the object in question available in the view.

The create method

To create a new fruit, we begin by navigating to the corresponding url, which renders a page with a form. In the form, the parameters are set and then sent to the `create` method in our controller, wherein the object is saved. The `save` method is inherited from **ActiveRecord Base** (follow the inheritance line of our model **fruit.rb**!), and is what adds a new fruit model to the database table. The method returns either `true` or `false` depending on its success. (The process is the same for `edit` and `update`!)

Try it out! What happens when running `Fruit.new` in the Rails console? What happens when running `Fruit.new(id: 100, user_id: 1, name: "Orange", is_moldy: true)`? Do these commands affect the outcome of `Fruit.all` (the contents of the database table)? Try instead running `Fruit.new(id: 100, user_id: 1, name: "Orange", is_moldy: true).save`, and see what happens :-).

redirect_to

We see that `create`, `update` and `delete` methods all have incorporated the `redirect_to` method. As the name suggests it redirects to a certain page. The first parameter points to the end destination and the second renders a flash message on the screen. These "end destinations" are accessed through the paths you see when running

```
rails routes | grep fruit
```

The edit path has `(@fruit)` at its end since the path is specific to each fruit, fsektionen.se/admin/frukter/[fruit_id]/redigera*, as one edits one fruit at a time. `admin_fruits_path` on the other hand refers to all fruits and takes you to fsektionen.se/admin/frukter*. Remember how we named this url path "frukter"? (Scroll up!)

fruit_params

Lastly, there is the `fruit_params` method. The contents of this method whitelists the attributes that are allowed to be saved, which is why this method is used in the `create` method. We have defined that `:fruit` is a required attribute, while the rest are optional. This was introduced to Rails as a security feature. `fruit_params` checks if `params[:id]` exist and then return a params hash with the given, accepted, attributes if it does.

The "regular" controller

This controller will be found in **web/app/controllers/fruits_controller.rb**. Since the explanations above were quite in-depth we'll write out the complete controller directly:

```ruby
class FruitsController < ApplicationController
  load_permissions_and_authorize_resource

  def index
    @fruits = initialize_grid(current_user.fruits)
  end

  def show
    @fruit = Fruit.find(params[:id])
  end
end
```

Through the `index` method we want to retrieve all the fruits that belong to the user. We are able to do so using the `includes` method as we already have defined the relation between a user and their fruits with Associations. Unlike the admin controller, there is the `show` method, which will be used to show a single fruit. *Is it necessary to define?*

Giving users abilities

In the top of the controller there is the peculiar line `load_permissions_and_authorize_resource`. This checks which user is logged in and if this user should be able to access the controller actions. It is therefore very important to not forget to add a line in `models/ability.rb` which grants the users the necessary privileges (or abilities). To allow the user to use the `index` and `show` actions add the following line to `ability.rb` in the end of the end of the block that checks wether a user id is present (which is the same as checking if a user is signed in):

```
can :read, Fruit
```

Writing `:read` here is equal to writing `[:index, :show]`.

You might wonder why this was not needed when defining the admin controller. This is because admins have all abilities available. There is therefore no need to add specific abilities, they are added automatically.

The API controller

This controller is located at **web/app/controllers/api/fruits_controller.rb**. The controller has a very similar structure to the other two but the logic in the functions differ a bit:

```
class Api::FruitsController < Api::BaseController
  load_permissions_and_authorize_resource

  def index
    @fruits = Fruit.all
    render json: @fruits,
    each_serializer: Api::FruitSerializer::Index
  end

  def show
    @fruit = Fruit.find(params[:id])
    render json: @fruit,
    serializer: Api::FruitSerializer::Show
  end
end
```

The requested objects are found exactly as in the **regular** controller through `Fruit.all` or `Fruit.find` but to then send it to the app or any other external source it needs to be formatted in a standardised manner. The format used for our API is json and to define how the formatting should be done and what attributes of the object to send, a serializer is used.

The serializer is located at **web/app/serializers/api/fruit_serializer.rb** and can define multiple formats for different use cases. We usually define these formats as subclasses with the same name as the method they are used in:

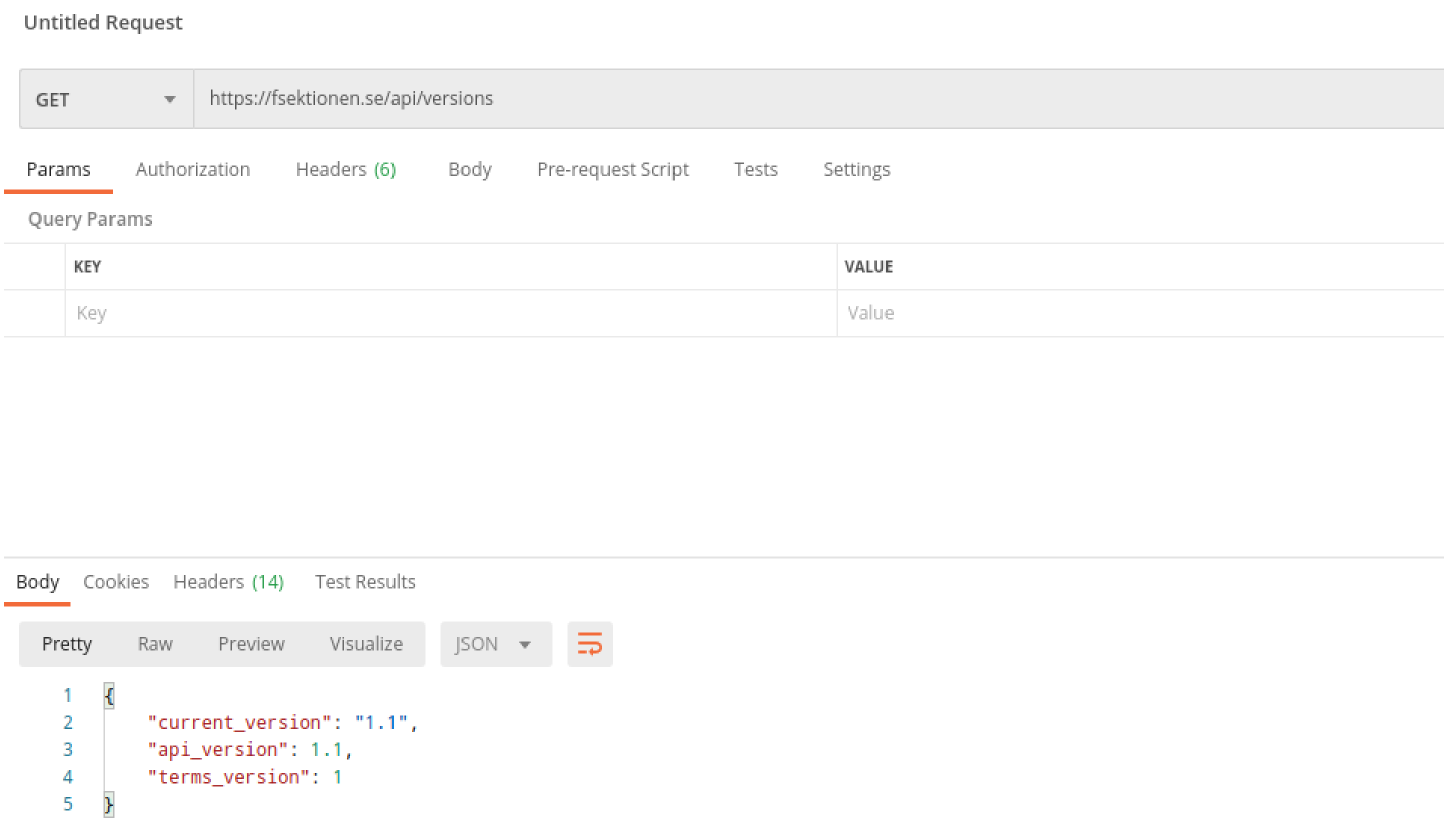
```
class Api::FruitSerializer < ActiveModel::Serializer

  class Api::FruitSerializer::Index < ActiveModel::Serializer
    attributes(:id, :name, :is_moldy)

    has_one :user

    class Api::UserSerializer < ActiveModel::Serializer
      attributes(:id, :firstname, :lastname)
    end
  end

  class Api::FruitSerializer::Show < ActiveModel::Serializer
    attributes(:id, :name, :user, :is_moldy)
  end
end
```

Testing the API

The API routes can be tested using Postman. This is a good habit to have since it enables you to ensure that the API responds in the way you expect it to.

Installing Postman is usually not a problem and a simple google search will likely give you the info you need.

Postman has an address bar in the upper part of the window where you type in the URL you wish to send a request to. For example, if you wish to know which api version our production server uses. Simply put *https://fsektionen.se/api/versions* in the address bar and select *GET* as method in the drop down menu to the left of the address bar. See the image below for reference (click for a larger version).



```
{
    "current_version": "1.1",
    "api_version": 1.1,
    "terms_version": 1
}
```

In the bottom half of the window is the response from the server which in this case is various versions. This specific endpoint does not required authentication and can be accessed by anyone. This is not the case for most endpoints where token authorization is handled by rails. One can retrieve a token with Postman by logging in through the login endpoint but there is an easier way to test endpoints if you are running a local server.

You might have seen the line `load_permissions_and_authorize_resource` which performs authentication and loads the current user. When testing locally, that line can be removed and the endpoint tested without having to login. Just be aware that any functionality having to do with the `current_user` will naturally not work since this variable is defined after login.

Design the views

The views have already been mentioned quite a lot. If things feel a bit confusing right now, the views might help with the understanding. The views are found in **web/app/views/** and define the HTML styling of the fruits' web pages.

Admin view

For the admin views we will create four files * index.html.erb * _form.html.erb * new.html.erb * edit.html.erb

which all previously have been referred to, one way or another.

Index

This page will list all existing fruits in a neat table. As we initialized the variable `@fruits = initialize_grid(Fruit)` in the `index` method, we can easily create a table using the grid method from the Ruby gem *wice_grid*.

```
<div class="col-md-12 col-sm-12 fruit-padding">
  <div class= "headline">
    <h2><%= title(t('.new')) %></h2>
  </div>
  <%= link_to(t('.new_fruit'), new_admin_fruit_path, class: 'btn primary')%>
</div>

<div class="col-md-12 col-sm-12 reg-page">
  <div class= "headline">
    <h1><%= title(t('.title')) %></h1>
  </div>
  <div class="col-md-12">
    <%= grid(@fruits, hide_submit_button: true, hide_reset_button: true) do␣
↪|g|
      g.column(name: t('.user'), attribute: 'user_id') do |fruit|
        fruit.user
      end

      g.column(name: t('.name')) do |fruit|
        fruit.name
      end

      g.column(name: t('.mold')) do |fruit|
        if fruit.is_moldy
          t('.moldy_yes')
        else
          t('.moldy_no')
        end
      end

      g.column(name: t('.edit')) do |fruit|
        link_to(t('.edit_fruit'), edit_admin_fruit_path(fruit),
                                  class: 'btn secondary')
      end

      g.column(name: t('.delete')) do |fruit|
        link_to(t('.delete_fruit'), admin_fruit_path(fruit),
                                    method: :delete,
```

(continues on next page)

(continued from previous page)

```
                                             data: {confirm: t('.confirm_delete')},
                                             class: 'btn danger')
      end
    end %>

    <button class="wg-external-submit-button btn primary" data-grid-name=
↪"grid">
      <%= t('wice_grid.filter') %>
    </button>

    <button class="wg-external-reset-button btn secondary" data-grid-name=
↪"grid">
      <%= t('wice_grid.reset_filter') %>
    </button>
  </div>
</div>
```

You will notice throughout these files that there are code snippets of the form `t('.something')`. These are used in place of regular strings, e.g. `"something"`, to enable Swedish and English translations. We will get to those later.

Code of the form `<%= code %>` are written in embedded Ruby, simply meaning we are embedding Ruby in an HTML file. An example of this is the aforementioned grid. In here we insert `@fruits` as a parameter, then for each row `g`, we define the content of each column. In the first column we put the fruit owner by calling `fruit.user`. This column is unique for having `attribute: 'user_id'` as a parameter, which will automatically render a field to filter the grid rows according to the user's name.

The third and fourth columns link to `edit` and `delete` for the fruit respectively. For the delete link we must have `method: :delete` as a parameter to specify that we want to delete the fruit. For `edit` we use the pre-existing path `edit_admin_fruit`. Here we can see why there is no need to define an `edit` method in the controller. In the index file there exists the `fruit` variable which points to each and every Fruit object. Through `edit_admin_fruit_path(fruit)` we make the fruit accessible in the edit view.

The last two sections render buttons used to filter and reset filter respectively. Above the grid we also have a button for creating a new fruit.

Form

The form is used both for creating and editing fruits. The form is what's called a *partial* since it will be rendered within other view files. This file is therefore named with an underscore, **_form.erb**.

```
<%= simple_form_for([:admin, fruit]) do |fruit|%>
    <%= fruit.input :user_id,
                    label: t('.user'),
                    collection: User.by_firstname.confirmed,
                    input_html: { class: 'select2' },
                    include_blank: true, prompt: t('.user_prompt'),
                    label_method: :print_email %>
    <%= fruit.input :name, label: t('.name')%>
    <%= fruit.input :is_moldy, label: t('.mold'), as: :radio_buttons %>
    <%= fruit.button :submit %>
    <%= link_to(t('.all'), admin_fruits_path, class: 'btn secondary') %>
<% end %>
```

Using `fruit.input` we define what attributes we would like the form to ask for. `fruit.button :submit` renders a button the user has to press when done. The button label depends on the occurring action; if the form is used to create a fruit it will say “Skapa Frukt”, or to edit a fruit it will say “Uppdatera Frukt”. Clicking this button runs either `create` or `update` in the controller. Next to it there will be a button which redirects back to the index page.

New

```
<div class="col-md-8 col-md-offset-2 col-sm-12 reg-page">
  <div class="headline">
    <h1> <%= t('.title')%> </h1>
  </div>

  <%= render 'form', fruit: @fruit %>
</div>
```

As mentioned, both **new** and **edit** will make use of the form. `fruit: @fruit` makes `@fruit` accessible in the form as the variable `fruit`.

Edit

```
<div class="col-md-8 col-md-offset-2 col-sm-12 reg-page">
  <div class="headline">
    <h1> <%= t('.title')%> </h1>
  </div>

  <%= render 'form', fruit: @fruit %>
</div>
```

This file is identical to **new**.

Styling with CSS

We implement CSS code in a new file **_fruits.scss** in **web/app/assets/stylesheets/partials/**.

```
.fruit-padding {
  margin-bottom: 35px;
}
```

In the index file we make use of `fruit-padding` in the very beginning. This simply adds a space below the top section.

User view

Here we will create the views for the regular user.

Index

Unlike the admin index view, we only list the fruits belonging to the current user. The grid will only have columns for the fruits’ names and moldiness. Additionally, we will link to each fruit’s show page.

```
<div class="col-md-12 col-sm-12 reg-page">
  <div class= "headline">
    <h1><%= t('.title')%></h1>
  </div>
  <div class="col-md-12">
    <%= grid(@fruits, hide_submit_button: true, hide_reset_button: true) do␣
↪|g|
      g.column(name: t('.name'), attribute: 'id') do |fruit|
        link_to(fruit.name, fruit_path(fruit))
      end

      g.column(name: t('.mold')) do |fruit|
        if fruit.is_moldy
                t('.moldy_yes')
        else
                t('.moldy_no')
        end
      end
    end %>

    <button class="wg-external-submit-button btn primary" data-grid-name=
↪"grid">
      <%= t('wice_grid.filter') %>
    </button>

    <button class="wg-external-reset-button btn secondary" data-grid-name=
↪"grid">
      <%= t('wice_grid.reset_filter') %>
    </button>
  </div>
</div>
```

Show

On the show page we simply render the fruit's name and a description of its moldiness.

```
<div class="col-md-12 col-sm-12 fruit-padding">
  <div class= "headline">
    <h2><%= title(t('.new')) %></h2>
  </div>
  <%= link_to(t('.new_fruit'), new_admin_fruit_path, class: 'btn primary')%>
</div>

<div class="col-md-12 col-sm-12 reg-page">
  <div class= "headline">
    <h1><%= title(t('.title')) %></h1>
  </div>
  <div class="col-md-12">
    <%= grid(@fruits, hide_submit_button: true, hide_reset_button: true) do␣
↪|g|
      g.column(name: t('.user'), attribute: 'user_id') do |fruit|
        fruit.user
      end

      g.column(name: t('.name')) do |fruit|
        fruit.name
```

(continues on next page)

(continued from previous page)

```
      end

      g.column(name: t('.mold')) do |fruit|
        if fruit.is_moldy
          t('.moldy_yes')
        else
          t('.moldy_no')
        end
      end

      g.column(name: t('.edit')) do |fruit|
        link_to(t('.edit_fruit'), edit_admin_fruit_path(fruit),
                                     class: 'btn secondary')
      end

      g.column(name: t('.delete')) do |fruit|
        link_to(t('.delete_fruit'), admin_fruit_path(fruit),
                                     method: :delete,
                                     data: {confirm: t('.confirm_delete')},
                                     class: 'btn danger')
      end
    end %>

    <button class="wg-external-submit-button btn primary" data-grid-name=
↪"grid">
      <%= t('wice_grid.filter') %>
    </button>

    <button class="wg-external-reset-button btn secondary" data-grid-name=
↪"grid">
      <%= t('wice_grid.reset_filter') %>
    </button>
  </div>
</div>
```

Write the translations

Every string we wish to be available in both English and Swedish, we have to define in files found in **web/config/locales**.

Views

```
sv:
  admin:
    fruits:
      index:
        title: Alla frukter
        name: Namn
        mold: Möglig?
        user: Ägare
        edit: Redigera
        edit_fruit: Redigera frukt
        delete: Radera
```

(continues on next page)

(continued from previous page)

```
        delete_fruit: Radera frukt
        confirm_delete: Vill du verkligen radera frukten?
        new: Skapa ny frukt
        new_fruit: Ny frukt
        moldy_yes: Ja
        moldy_no: Nej
      edit:
        title: Redigera frukt
      new:
        title: Ny frukt
      form:
        name: Namn
        mold: Möglig?
        user: Ägare
        user_prompt: Välj användare
        all: Alla frukter
```

```
sv:
  fruits:
    index:
      title: Dina frukter
      name: Namn
      mold: Möglig?
      moldy_yes: Ja
      moldy_no: Nej
    show:
      moldy: Den här frukten är möglig.
      not_moldy: Den här frukten är fräsch.
```

```
---
en:
  admin:
    fruits:
      index:
        title: All fruits
        name: Name
        mold: Moldy?
        user: Owner
        edit: Edit
        edit_fruit: Edit fruit
        delete: Delete
        delete_fruit: Delete fruit
        confirm_delete: Are you sure you want to delete this fruit?
        new: Create new fruit
        new_fruit: New fruit
        moldy_yes: "Yes"
        moldy_no: "No"
      edit:
        title: Edit fruit
      new:
        title: New fruit
      form:
        name: Name
        mold: Moldy?
        user: Owner
        user_prompt: Choose user
```

(continues on next page)

(continued from previous page)

```
        all: All fruits
```

```
---
en:
  fruits:
    index:
      title: Your fruits
      name: Name
      mold: Moldy?
      moldy_yes: "Yes"
      moldy_no: "No"
    show:
      moldy: This fruit is moldy.
      not_moldy: This fruit is fresh.
```

Model

```
sv:
  activerecord:
    models:
      fruit:
        one: Frukt
        other: Frukter
    attributes:
      fruit:
        user: Användare
        name: Namn
        is_moldy: Möglig?
```

```
---
en:
  activerecord:
    models:
      fruit:
        one: Fruit
        other: Fruits
    attributes:
      fruit:
        user: User
        name: Name
        is_moldy: Moldy?
```

Creating menus to access the fruits

Regular menu

Adding a new menu is actually done on the website. Sign in as an admin and head to *administrera/menyer*. Now add a new menu with link `/frukter` and other appropriate attributes. The sorting index decides in which order to show the menus, omitting this field will put the link in the bottom of the menu. The column decides which of the columns in the menu to put the link.

Admin dropdown menu

Lastly, we would like there to be a link to the admin pages in the dropdown menu. We go to **web/app/views/layouts/dropdowns/_admin_dropdown.html.erb** and add `Fruit` to the `Övrigt` section.

```
<% all_privileges = { 'Användare' => [User, MailAlias, Group, Permission],
                      'Poster' => [Election, Council, Document],
                      'Information' => [News, Event, BlogPost],
                      'Spindelman' => [Menu, ShortLink, Constant, Category],
                      'Pryl' => [Rent, Tool, Door, AccessUser, Key],
                      'Övrigt' => [CafeShift, Introduction, Adventure,␣
↪WorkPost, Faq, Page, Album, Contact, Meeting, Song, Fruit] } %>
```

Read more

- Validations
- Associations
- Routes
- Controllers
- Inheritance in Rails Controllers
- Rails rendering
- Parameters

6.2 App examples

Here are some examples of app code.

6.2.1 Fruits App

Introduction

This is a simple example illustrating the approximate workflow for making changes to the app. It will build on the fruits example from our web examples, and we will implement a basic page displaying the fruits in the app. This is not a comprehensive introduction to dart or flutter, and the example assumes that you are at least somewhat familiar with the basics. I recommend checking out the official flutter documentation for an introduction to the framework, they do a much better job explaining the fundamentals than I ever will. The main point of the example is to show how we interact with the website in the app through our API.

Preparation

Preparing your local web

We will run the app against our local web and not stage in this example, and we need a working implementation of "Fruits" from our web examples. If you want, you can implement that yourself before you continue, or you can checkout the johanna-fruits-example branch on fsek/web. You probably want to do

```
git pull origin master
```

if you use the johanna-fruits-example branch, since quite a few dependency upgrades have been deployed since the example was written. Also make sure that you have read through the example and understand roughly how it works, we will assume that you are somewhat familiar with the backend for “fruits” from this point onwards. Make sure that foreman s works, and create a few fruits so we have some data to work with.

App preparations

Make sure that you have cloned the App2 repository somewhere, and that you can run the app on an emulator.

Create a git branch

Once you have made sure that you can run the web server, we can start working on the app. As always when we make code changes, the first thing we want to do is to create a new branch. Head over to the location of your App2 repo clone, and do

```
git checkout -b "app-fruit-example"
```

ofc you can use whatever name you like for the branch, but something descriptive is often nice.

Configure the app to run against localhost

Normally, the app will fetch data from stage.fsektionen.se, but we want to use the fruits-api which we are currently developing locally. Navigate to **lib/environments/environment.dart**. It will look like this:

```
class Environment {
  static const String API_URL = "https://fsektionen.se";
  static const String CABLE_URL = "wss://fsektionen.se/cable";
}
```

We want to change the API_URL variable to use localhost:3000, like this:

```
static const String API_URL = "http://10.0.2.2:3000"
```

10.0.2.2 is the adress for localhost on the android emulator. We do not care about the CABLE_URL, since we will not use web sockets in this example. Start your local web server and make sure that the app connects to it from the emulator.

Adding an empty page for fruits

Now we are going to add an empty place holder page where we can eventually put our fruits. We will put it under the “Hilbert Cafe” tab in the “Övrigt” section. First, we create the base files for what will eventually become our fruits page. Create a new directory under **/lib/screens** called **fruits**, and add the file **fruits.dart**. We put a basic empty stateful widget in it:

```
import 'package:flutter/material.dart';

class FruitPage extends StatefulWidget {
  const FruitPage({Key? key}) : super(key: key);

```

(continues on next page)

(continued from previous page)

```
  @override
  _FruitPageState createState() => _FruitPageState();
}

class _FruitPageState extends State<FruitPage> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Frukter'),
      ),
    );
  }
}
```

Now we will need to be able to access our new page somehow. We want to add it among the items in "Övrigt", so we head over to **lib/screens/other.dart**. Among the import statements at the top of the file, we add:

```
import 'package:fsek_mobile/screens/fruits/fruits.dart';
```

so we can use our fruit page. Around line 20, we find the line

```
final catagories = ["Sångbok", "Bildgalleri", "Hilbert Café"];
```

Add the string "Frukter" to the end of the list. Further down in the file, we find a routeMap. Here we add the key-value pair

```
"Frukter": FruitPage()
```

Now we should see "Frukter" under "Hilbert Cafe" in "Övrigt" in the app, clicking on it should open an empty page with an appbar saying Frukter.

Adding the fruits

So far, we have only added some dummy static content. Now, we get to the interesting part! It is time to bring in our fruits! We are going to interact with the website using the API. On the web-side of things, we create actions in the API-controller that specify how the web server will handle different kinds of requests. These will typically be standard rails actions that might sound familiar, such as "index" or "show". But how do we tell the server to perform these actions? Through rails-magic (or routes we manually specify for special actions), these actions are mapped to different HTTP-requests. When we send the correct HTTP-requests, the server will perform the corresponding action and typically respond by sending us some data in JSON-format. We then parse that json object, and do something useful with it. If you wonder what actions are available for a certain object, or what request you use to perform that action you can head over to the location of your web repository, and do for example

```
rails routes | grep fruits
```

if you are interested in the available actions and corresponding requests for fruits. (Sidenote: You can use Postman to communicate with the API outside of app development. This is very useful to test what kind of responses you get from different actions, and make it easier to figure out what you will need to implement in the app.)

Control question: What actions are available through the API for our fruits? What actions are available for events?

To summarize, the general idea for piping data from the web to the app is as follows: We create a *service* that will make an http request to the server, corresponding to an action in the API-controller over on our rails web site. The server will respond with a json object containing the data we requested. We then parse this raw json-data into a dart object that we can use in our app.

Adding a fruit model and JSON parser

Sending data over API:s as JSON-objects is very common, so we get a lot of help from the app framework when we want to parse these. In order to represent a fruit as a dart object, we create a fruit model. This is in many ways similar to the model for fruit we created in rails. We head on over to **lib/models/home** and create the file **fruit.dart** (this is not rails, we can name our model whatever we want, but descriptive names are nice). We are also going to need a file **fruituser.dart** in the same folder, for reasons that will soon be explained.

```
import 'package:json_annotation/json_annotation.dart';
import 'fruituser.dart';

part 'fruit.g.dart';

@JsonSerializable()
class Fruit {
  int? id;
  String? name;
  bool? is_moldy;
  FruitUser? user;

  Fruit();

  factory Fruit.fromJson(Map<String, dynamic> json) => _$FruitFromJson(json);

  Map<String, dynamic> toJson() => _$FruitToJson(this);
}
```

```
import 'package:json_annotation/json_annotation.dart';
import 'fruituser.dart';

part 'fruit.g.dart';

@JsonSerializable()
class Fruit {
  int? id;
  String? name;
  bool? isMoldy;
  FruitUser? user;

  Fruit();

  factory Fruit.fromJson(Map<String, dynamic> json) => _$FruitFromJson(json);

  Map<String, dynamic> toJson() => _$FruitToJson(this);
}
```

Don't worry about the warnings your linter is giving you, we will soon auto-generate a bunch of stuff, but first some comments on the contents of these files. We want our fruit class to contain the fruit attributes that is sent to us over the API. The serializer is responsible for converting the rails fruit object into json format, so we peek at the fruit serializer over in our web repository (if you do not have such a serializer, read the prerequisits again). If you have the same implementation as the one on "johanna-fruits-example", we see that we send the attributes id, name and is_moldy. These explain the first three attirbutes in our **fruit.dart** file. We also see

```
has_one :user
```

in the serializer. This means that we send the associated "user" object as a nested json in our fruit json representation. When sending the corresponding user for a fruit, we do not need, nor want, every bit of information about that user, so the fruit serializer for the index action has a UserSerializer that tells us which attributes to send for the user that owns the fruit. We can see that we only send id, firstname and lastname. We thus want a model for the type of user that is sent by the fruit-API, and we call it FruitUser. FruitUser is a user that only has an id, a firstname and a lastname. We will now auto-generate the code for parsing json into Fruits and FruitUsers. To do this, we run

```
flutter pub run build_runner build
```

in the command line (this is my favorite command of all time). This should create files like **fruit.g.dart**.

Creating a FruitService

Now that we have a model for our fruits and can parse them from json, we need to create a service that makes the correct http request to the server, recieves the json response, parses it to dart objects and returns the results. We will start with a service for the basic "index" action in rails. In **lib/services/** make a file called **fruit.service.dart**

```
import 'abstract.service.dart';
import 'package:fsek_mobile/models/home/fruit.dart';

class FruitService extends AbstractService {
  Future<List<Fruit>> getFruits() async {
    Map json = await AbstractService.get("/frukter");
    List<Fruit> fruitlist =
        (json['fruits'] as List).map((data) => Fruit.fromJson(data)).
↪toList();
    return fruitlist;
  }
}
```

AbstractService wraps the basics for making an http request. We do not need to worry too much about it here, but feel free to take a look at it if you want to. the index action in ruby will gives an object of the form

```
{"fruits" : [*JSON representation of fruit 1*, ....]}
```

and we parse that into a list of fruits. Depending on internet and server speed etc, we might have to wait a while for the response to come, which async/await handles: we make a promise that our object will arrive at some point in the future, and can continue execution of the program in the meantime. Finally, we need to register our FruitService on the service locator so that we can use it. Go to **lib/services/service_locator.dart** and add the import statement

```
import 'package:fsek_mobile/services/fruit.service.dart';
```

and

```
locator.registerLazySingleton(() => FruitService());
```

where appropriate.

Adding the fruits on our page

Now we have a way to fetch fruits from the website, so it's time to return to our mostly empty fruit page and start to populate it. Go back to **lib/screens/fruits/fruits.dart**, and modify it to look like this:

```
import 'package:flutter/material.dart';
import 'package:fsek_mobile/models/home/fruit.dart';
import 'package:fsek_mobile/models/home/fruituser.dart';
```

(continues on next page)

(continued from previous page)

```dart
import 'package:fsek_mobile/services/fruit.service.dart';
import 'package:fsek_mobile/services/service_locator.dart';

class FruitPage extends StatefulWidget {
  const FruitPage({Key? key}) : super(key: key);

  @override
  _FruitPageState createState() => _FruitPageState();
}

class _FruitPageState extends State<FruitPage> {
  List<Fruit>? fruits;
  void initState() {
    locator<FruitService>().getFruits().then((value) => setState(() {
          this.fruits = value;
        }));
    super.initState();
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Frukter'),
      ),
    );
  }
}
```

We have now added an initState function for our stateful widget. This initializes the state of the widget when it is first built. We add the attribute fruits, which will contain a list of all the fruit objects we recieve from the API call, and make the API call with the service we created. With our list of fruits ready, we can begin to fill the page. We add a listView with text widgets containing the fruit names to the page.

```dart
import 'package:flutter/material.dart';
import 'package:fsek_mobile/models/home/fruit.dart';
import 'package:fsek_mobile/models/home/fruituser.dart';
import 'package:fsek_mobile/services/fruit.service.dart';
import 'package:fsek_mobile/services/service_locator.dart';

class FruitPage extends StatefulWidget {
  const FruitPage({Key? key}) : super(key: key);

  @override
  _FruitPageState createState() => _FruitPageState();
}

class _FruitPageState extends State<FruitPage> {
  List<Fruit>? fruits;
  void initState() {
    locator<FruitService>().getFruits().then((value) => setState(() {
          this.fruits = value;
        }));
    super.initState();
  }
```

(continues on next page)

(continued from previous page)

```
  @override
  Widget build(BuildContext context) {
    if (fruits == null) {
      return Scaffold(
        appBar: AppBar(
          title: const Text('Frukter'),
        ),
      );
    } else {
      return Scaffold(
        appBar: AppBar(
          title: const Text('Frukter'),
        ),
        body: ListView(
            children: [...fruits!.map((fruit) => Text(fruit.name ?? ""))]),
      );
    }
  }
}
```

This is a good point to stop, take a step back, and make sure that everything is working as intended. We should now be able to click on the "Frukter" button in "Övrigt", and see an (ugly) list of fruit names, with all the fruits we have created on the web.

Improving the list of fruits

Once we've made sure that things work as intended up to this point, it is time to start doing some basic styling. We probably want the list of fruits to contain clickable cards, that direct you to individual fruit pages with slightly more information in them. Eventually, that means adding the "show" action to our FruitService, but before we do that, we prepare the cards. We do something similar to this:

```
import 'package:flutter/material.dart';
import 'package:fsek_mobile/models/home/fruit.dart';
import 'package:fsek_mobile/models/home/fruituser.dart';
import 'package:fsek_mobile/services/fruit.service.dart';
import 'package:fsek_mobile/services/service_locator.dart';

class FruitPage extends StatefulWidget {
  const FruitPage({Key? key}) : super(key: key);

  @override
  _FruitPageState createState() => _FruitPageState();
}

class _FruitPageState extends State<FruitPage> {
  List<Fruit>? fruits;
  void initState() {
    locator<FruitService>().getFruits().then((value) => setState(() {
          this.fruits = value;
        }));
    super.initState();
  }

  @override
  Widget build(BuildContext context) {
```

(continues on next page)

(continued from previous page)

```
    if (fruits == null) {
      return Scaffold(
        appBar: AppBar(
          title: const Text('Frukter'),
        ),
      );
    } else {
      return Scaffold(
        appBar: AppBar(
          title: const Text('Frukter'),
        ),
        body:
            ListView(children: [...fruits!.map((fruit) => _
↪FruitCard(fruit))]),
      );
    }
  }
}

/*
 * note that this widget is stateless. We only want to set the fruit
 * for the widget once and never change it, so we use final. It will
 * never change state after construction, so it is stateless and not stateful
 */
class _FruitCard extends StatelessWidget {
  final Fruit fruit;

  @override
  _FruitCard(this.fruit);

  Widget build(BuildContext context) {
    return SizedBox(
      height: 50,
      child: Card(
        child: InkWell(
          onTap: () {},
          child: Align(
            alignment: Alignment.center,
            child: Text(fruit.name!, style: TextStyle(fontSize: 20)),
          ),
        ),
      ),
    );
  }
}
```

We've replaced the text widget in our list view with a custom private widget, _FruitCard, that creates clickable cards for each fruit. The click does not do anything yet: the onTap function is an empty lambda expression. Feel free to expermient with the styling of the _FruitCard here. Perhaps we want a different color? A different text font? Different layouts?

Adding pages for each fruit

Modify service

With an (hopefully pretty) fruit list completed, we can think about adding individual pages for each fruit. First, we modify our service to fetch data for singular fruits. This is the "show" action in rails, and we reach that action in the API-controller by sending a http GET request to the path API_URL/frukter/*id for fruit we want*. Go vack to **lib/services/fruit.service.dart** and add a function getFruit:

```
Future<Fruit> getFruit(int id) async {
    Map json = await AbstractService.get("/frukter/$id");
    return Fruit.fromJson(json['fruit']);
}
```

Adding a widget to display individual fruits

We will now make a very basic widget to display the fruits individually, to make sure everything works. Create a new file in **lib/screen/fruits**, and call it for example **fruitview.dart** (again, reminder that this is not rails and we can name things whatever we want).

```
import 'package:flutter/cupertino.dart';
import 'package:fsek_mobile/services/fruit.service.dart';
import 'package:fsek_mobile/services/service_locator.dart';

import 'package:flutter/material.dart';
import 'package:fsek_mobile/models/home/fruit.dart';

class FruitView extends StatefulWidget {
  final int id;
  const FruitView({Key? key, required this.id}) : super(key: key);

  @override
  _FruitViewState createState() => _FruitViewState();
}

class _FruitViewState extends State<FruitView> {
  Fruit? fruit;
  void initState() {
    locator<FruitService>().getFruit(widget.id).then((value) => setState(() {
          this.fruit = value;
        }));
    super.initState();
  }

  @override
  Widget build(BuildContext context) {
    if (fruit == null) {
      return Scaffold(
        appBar: AppBar(
          title: const Text('Frukt'),
        ),
      );
    } else {
      return Scaffold(
        appBar: AppBar(
          title: const Text('Frukt'),
        ),
```

(continues on next page)

(continued from previous page)

```
        body: Text(fruit!.name ?? ""),
      );
    }
  }
}
```

Compare this to what we did initially when we made the simple text list for the fruits. This time, we get a single fruit instead of a list of fruits, but otherwise the basic structure for the widgets are very similar.

Finally, we want to make the cards in our list navigate to this widget. Replace the empty onTap in the InkWell in the _FruitCard with

```
onTap: () {
    Navigator.push(
        context,
        MaterialPageRoute(
            builder: (context) => FruitView(id: fruit.id ?? -1)));
},
```

What import statement will you need to add for this to work? Make sure that the page navigation works when you click on the _FruitCards.

Styling the individual fruit page

Finally, we add some basic styling to the individual fruit pages. For example, we probably want to display the moldiness of the fruits on the individual page in some way. Here's an example for how one could modify **lib/screens/fruits/fruitview.dart** to be somewhat more helpful

```
import 'package:flutter/cupertino.dart';
import 'package:fsek_mobile/services/fruit.service.dart';
import 'package:fsek_mobile/services/service_locator.dart';

import 'package:flutter/material.dart';
import 'package:fsek_mobile/models/home/fruit.dart';

class FruitView extends StatefulWidget {
  final int id;
  const FruitView({Key? key, required this.id}) : super(key: key);

  @override
  _FruitViewState createState() => _FruitViewState();
}

class _FruitViewState extends State<FruitView> {
  Fruit? fruit;
  void initState() {
    locator<FruitService>().getFruit(widget.id).then((value) => setState(() {
          this.fruit = value;
        }));
    super.initState();
  }

  @override
  Widget build(BuildContext context) {
    if (fruit == null) {
```

(continues on next page)

(continued from previous page)

```
      return Scaffold(
        appBar: AppBar(
          title: const Text('Frukt'),
        ),
      );
    } else {
      return Scaffold(
        appBar: AppBar(
          title: const Text('Frukt'),
        ),
        body: _FruitWidget(this.fruit!),
      );
    }
  }
}

class _FruitWidget extends StatelessWidget {
  final Fruit fruit;
  String moldStatus = "";
  @override
  _FruitWidget(this.fruit) {
    /* We are optimistic. If is_moldy is null we assume the fruit is fresh */
    if (this.fruit.is_moldy ?? false) {
      this.moldStatus = "möglig";
    } else {
      this.moldStatus = "ej möglig";
    }
  }

  Widget build(BuildContext context) {
    return SizedBox(
      height: 50,
      child: Card(
        child: Align(
          alignment: Alignment.center,
          child: Text("${fruit.name!}: ${moldStatus} ",
              style: TextStyle(fontSize: 20)),
        ),
      ),
    );
  }
}
```

Once again, feel free to experiment here.

Ideas to try on you own

This example covers the basics of how to fetch data from the web and use it to implement new features in the app. There are a lot of things you could try to do on your own to expand upon it if you want to learn more about how the web and app interacts! Here are some ideas:

1. Currently, the fruit index lists all fruits for all users. We probably only want to fetch the fruits belonging to the current user. How would you fix that in the web backend?
2. It would be nice to be able to create and delete fruits in the app! What actions would you have to add to the API-controller? What http requests to does actions correspond to? What would you add to the fruit service to

carry out those actions?

CHAPTER 7

Spider Conference

The Spider conference is an annual conference for the Spidermans of the F-guild. The conference is also known as *Torna Championship in Programming (TCP)*. The idea of the conference is to arrange a weekend with lectures, team building activities and a programming competition.

7.1 Spider Conference 2019

The task for this year's TCP (2019) was to create an online webshop for the F-guild. Necessary preparations should be done in the backend to fetch new products and display them in the app where it also should be possible to purchase the product. The store should consist of atleast 5 products with name, price and a picture.

Below one can find a detailed description of the task as well as an example solution.

7.1.1 Task

The task for the second TCP (2019) was to create an online webshop for the F-guild. Below one can find a detailed description of the task.

Rails

1. Create a database table `store_product` in `web/db/migrate`. Each product should have the following fields: `name`, `price`, `image_url` and `in_stock`. Remember to assign each field with an appropriate type, specify whether or not it should be mandatory and or if it should have a default value. The price should be saved in Swedish öre. Use the code below and add the missing code.

```
# web/db/migrate/YYYYMMDDHHMMSS_create_store_products.rb
class CreateStoreProducts < ActiveRecord::Migration[5.0]
  def change
    create_table :store_products do |t|
      # TODO. Add the four fields here.
```

(continues on next page)

(continued from previous page)

```
      t.timestamps null: false
    end
  end
end
```

2. Create the corresponding `model` for Rails in `web/app/models`. Think of the validations, e.g. that a price should not be able to be negative. Also, implement a `to_s` method in the `model`.
3. Migrate the database using `rails db:migrate` and create a few test products using the Rails console. This console in entered by typing `rails c`. *Hint:* By typing `StoreProduct` in the Rails console you should be able too see all of its fields. This is a good way to test that everything works as expected.
4. Add an admin route to `web/config/routes.rb` by adding a few lines in the "User-related routes" section. The path to the products should be `/produkter`. You have successfully set up the route if you get an `uninitialized constant` error when going to `tcp://localhost:3000/admin/produkter`.
5. Implement an admin `controller`. Copy and paste the following script and fill in the missing code:

```
# web/app/controllers/admin/store_products_controller.rb
class Admin::StoreProductsController < Admin::BaseController
  load_permissions_and_authorize_resource

  def new
    # TODO
  end

  def index
    # TODO. This method should return a grid initialization.
  end

  def edit
    @store_product = StoreProduct.find(params[:id])
  end

  def create
    # TODO. First, make a new StoreProduct object and then try to save it.
    # Use `redirect_to` with an appropriate path and `notice` saying␣
↪something
    # relevant depending on if it was saved successfully or not.
  end

  def update
    # TODO. The StoreProduct that is being updated should be named @store_
↪product.
  end

  def destroy
    @store_product = StoreProduct.find(params[:id])
    if @store_product.destroy
      redirect_to admin_store_products_path, notice: alert_
↪destroy(StoreProduct)
    else
      redirect_to edit_admin_store_product_path, notice: alert_danger(
↪'Kunde inte förinta produkt')
    end
  end
```

(continues on next page)

(continued from previous page)

```
  # All methods below this will be private
  private

  def store_product_params
    # This method ensures that the hash we get from the view has
    # a `StoreProduct`, and that we allow the permitted fields to be␣
↪updated.
    params.require(:store_product).permit(:name, :price, :image_url, :in_
↪stock)
  end
end
```

Hints: Type `rails routes | grep store` to see all paths that have the word `store` in it. Make use of the `store_product_params` method when creating a new product.

6. Create a few basic `views` to list and create new products. These `views` should be placed in `web/views/admin/store_products`, namely

```
web/views/admin/store_products/
    _form.html.erb
    edit.html.erb
    index.html.erb
    new.html.erb
```

The code for `edit.html.erb` can be found here:

```
<% # web/app/views/admin/store_products/edit.html.erb %>
<div class="col-md-10 col-md-offset-1 col-sm-12 reg-page">
  <div class="headline">
    <h1><%= 'Redigera produkt' %></h1>
  </div>
  <%= render('form', store_product: @store_product) %>
  <hr>
  <%= link_to('Förinta', admin_store_product_path(@store_product),
                              method: :delete,
                              data: {confirm: 'Är du säker på att du vill␣
↪förinta produkten?'},
                              class: 'btn danger pull-right') %>
  <%= link_to('Alla produkter', admin_store_products_path, class: 'btn␣
↪secondary') %>
</div>
```

When you have implemented the views, make sure that they work as expected before moving to the next task.

7. Create a `serializer` for the products. Copy and paste the following script and implement the missing code:

```
# web/app/serializers/api/store_product_serializer.rb
class Api::StoreProductSerializer < ActiveModel::Serializer
  class Api::StoreProductSerializer::Index < ActiveModel::Serializer
    # TODO. Include all fields
  end
end
```

8. Create an `API controller` for the store and implement the `index` method below. *Hint:* By doing task 10 and commenting out `load_permissions_and_authorize_resource` you can test if your `API controller` and `serializer` works as expected.

```
# web/app/controllers/api/store_products_controller.rb
class Api::StoreProductsController < Api::BaseController
  load_permissions_and_authorize_resource

  def index
    # TODO. This should return a JSON object containing all products.
  end
end
```

9. Add the rights to fetch the products for all signed in users in `abilities`. This file can be found in `web/app/models/ability.rb`.
10. Add an API route for the created `API controller` in `routes.rb` and test that it works. The path will become what you write after `resources`, e.g. `tcp://localhost:3000/api/songs`. *Hint:* By removing `load_permissions_and_authorize_resource` from the `API controller` you can fetch the data without being logged in, allowing you to simply test your `API controller` and `serializer`.

App

0. Replace line 9 in `app/www/index.html` with:

```
<!-- app/www/index.html -->
<meta http-equiv="Content-Security-Policy" content="default-src 'self'␣
↪https://stage.fsektionen.se https://fsektionen.se wss://fsektionen.se␣
↪wss://stage.fsektionen.se gap://ready 'unsafe-inline' 'unsafe-eval';␣
↪style-src 'self' 'unsafe-inline'; child-src 'self' https://www.youtube.
↪com gap://ready; media-src *; img-src * 'self' https://stage.fsektionen.
↪se https://fsektionen.se data:" />
```

1. Create a HTML, SCSS and JS file for the F-store. Don't forget to load the JS file in `index.html` and the SCSS file in `index.scss`. Note that the JS file cannot be loaded before a few basic JS files have been loaded. Copy and paste the following outline of the JS file to the newly created one:

```
// app/www/js/store.js
$$(document).on('', '', function () {        // TODO
  let storeProductAPIEndpointURL = '';       // TODO

  $.getJSON(storeProductAPIEndpointURL)
    .done(function(resp) {
      initStore(resp);
    })
    .fail(function(resp) {
      console.log(resp.statusText);
    });

  function initStore(resp) {
    // TODO
  }
});
```

   This is the general structure of our JS files. We first fetch the data and then send it to a function called `initSomething` where we handle the rest. Note that the other files don't need to contain any code at this point.

2. Add routes to the store in the alternatives view. The `name` should be `store` and `url` should be the relative path to the created HTML file. The routes are defined in `index.js`.

3. Implement the created HTML file similarly to the other pages. Remember to set the `date-name` to the `name` you defined in the routes in the previous task, i.e. `store`.
4. Add navigation to the new page in the alternatives tab. This is done by adding a few lines of code to the `<div id="view-alternatives" class="view tab"></div>` in `index.html`. The `<a>` tag should referene to the path you defined in the routes in task 2.
5. Catch the `page:init` event in the created JS file. Make sure that it works by logging `Spodermon iz kewl`.
6. Fetch data from the API endpoint called `store_products` and log it.
7. Create a template in `index.html` and test that it works. The latter is done by calling the template in the JS file and storing the HTML code in the store container. Note that the template does not have to handle potential input data, it should only be able to be used correctly.
8. Edit the template such that it generates a store. The page should make use of Framework7 Cards to display information about the products. On the cards there should be a button where one should be able to purchase the product. Each button should have a data attribute where the `id` of the product is stored. In this case the attribute will be named `data-id`. Remember that the price of the products are in Swedish Öre. *Hint:* Scroll down on the Framework 7 Cards documentation to see some examples.
9. Catch the `click` event when a buy button has been clicked using the jQuery method `on()`. This code should be placed inside the `initStore` function. When the button has been clicked, it should make a `POST` request to `https://stage.fsektionen.se/api/store_orders`. This should be done by calling the following function with the product `id`, which one can get from the data attribute of the button, as the input argument:

```
// app/www/js/store.js
function buyProduct(id) {
  $.ajax({
    url: '',                    // TODO
    type: '',                   // TODO
    dataType: 'json',
    data: {},                   // TODO
    success: function(resp) {
      app.dialog.alert(resp.success, 'Varan är köpt');
    },
    error: function(resp) {
      app.dialog.alert(resp.responseJSON.error);
    }
  });
}
```

   Make sure that the `POST` request is successful. The data of the `POST` request should contain an `item` object with `id` and `quantity` as data. Below you can find an example of the structure of the data.

```
"item": {
    "id": 1,
    "quantity": 1
}
```

10. Style the page so it looks fresh.

7.1.2 Solution

This is an example implementation of the F-store. It can also be found in its entirety on GitHub here: Rails, app.

Rails

1. The time at which the file was created and its class name should be in the filename.

   ```
   # web/app/db/migrate/YYYYMMDDHHMMSS_create_store_products.rb
   class CreateStoreProducts < ActiveRecord::Migration[5.0]
     def change
       create_table :store_products do |t|
         t.string :name, null: false
         t.integer :price, null: false, default: 0
         t.text :image_url
         t.boolean :in_stock, null: false

         t.timestamps null: false
       end
     end
   end
   ```

   In this example implementation we have set the fields `name`, `price` and `in_stock` as mandatory. This was done by specifing `null:  false`. We have also set the default `price` to zero using `default:  0`. Thus one can create a product without specifying a price, since the default value then will be set, resulting in the field being filled. We also always want to have `timestamps` to be able to see when each product was created.

2. For Rails to recognize the model it is important that the class and filename is in singular.

   ```
   # web/app/models/store_product
   class StoreProduct < ApplicationRecord
     validates :name, presence: true
     validates :price, numericality: { greater_than_or_equal_to: 0 }

     scope :in_stock, -> { where(in_stock: true) }

     def to_s
       name
     end
   end
   ```

   Here we require that each `StoreProduct` must have a `name` by setting `presence:  true`. In the `model` one can also specify `scopes`. In this case, calling `StoreProducts.in_stock` will return all the products that have the attribute `in_stock` set to `true`.

3. Creating and saving a `StoreProduct` can e.g. by done by typing

   ```
   StoreProduct.create!(name: 'Product 1', price: 100, in_stock: true)
   ```

   into the Rails console.

4. Here, the admin path to the products will become `/admin/store_products`. The `except` statement can be used if some methods are not implemented in the `controller`, which in this case is the `show` action.

   ```
   # web/app/config/routes.rb
   namespace :admin do
     resources :store_products, except: [:show], path: :produkter
   end
   ```

5. Here follows the entire `controller` file:

```ruby
# web/app/controllers/admin/store_products_controller.rb
class Admin::StoreProductsController < Admin::BaseController
  load_permissions_and_authorize_resource

  def new
    @store_product = StoreProduct.new
  end

  def index
    @store_products = initialize_grid(StoreProduct.all, order: :name)
  end

  def edit
    @store_product = StoreProduct.find(params[:id])
  end

  def create
    @store_product = StoreProduct.new(store_product_params)
    if @store_product.save
      redirect_to admin_store_products_path, notice: alert_
↪create(StoreProduct)
    else
      redirect_to new_admin_store_product_path(@store_product), notice:␣
↪alert_danger('Kunde inte skapa produkt')
    end
  end

  def update
    @store_product = StoreProduct.find(params[:id])
    if @store_product.update(store_product_params)
      redirect_to admin_store_products_path, notice: alert_
↪update(StoreProduct)
    else
      redirect_to edit_admin_store_product_path(@store_product), notice:␣
↪alert_danger('Kunde inte uppdatera produkt')
    end
  end

  def destroy
    @store_product = StoreProduct.find(params[:id])
    if @store_product.destroy
      redirect_to admin_store_products_path, notice: alert_
↪destroy(StoreProduct)
    else
      redirect_to edit_admin_store_product_path, notice: alert_danger(
↪'Kunde inte förinta produkt')
    end
  end

  private

  def store_product_params
    params.require(:store_product).permit(:name, :price, :image_url, :in_
↪stock)
  end
end
```

6. Here follows the code for all the `views`:

```
<% # web/app/views/admin/store_products/index.html.erb %>
<div class="headline">
  <h1><%= title('Produkter') %></h1>
</div>

<div class="col-md-2 col-sm-12">
  <%= link_to('Ny produkt', new_admin_store_product_path, class: 'btn␣
↪primary') %>
</div>

<div class="col-md-10 col-sm-12">
  <%= grid(@store_products) do |g|
    g.column(name: 'Namn', attribute: 'name') do |product|
      link_to(product, edit_admin_store_product_path(product))
    end
    g.column(name: 'Pris', attribute: 'price', filter: false)
    g.column(name: 'I lager', attribute: 'in_stock', filter: false) do␣
↪|product|
      if product.in_stock? then t('global.yes') else t('global.no') end
    end
  end -%>
</div>
```

Two comments regarding the code above. Firstly, the `filter:  false` argument will remove the possibility to search that column, i.e. that one cannot search for all prodcuts with e.g. the price `37`. Secondly, for the `in_stock` column we replace the value with `t('global.yes')` or `t('global.no')` depending on if the product is in stock or not. Rails fetches these values from a translation file (`web/config/locales/views/global.sv.yml` if the website is set to display in Swedish) where a (Swedish) translation of `Yes` and `No` exists.

```
<% # web/app/views/admin/store_products/_form.html.erb %>
<%= simple_form_for([:admin, store_product]) do |f| %>
  <%= f.input :name %>
  <%= f.input :price %>
  <%= f.input :in_stock %>
  <%= f.input :image_url %>
  <%= f.button :submit %>
<% end %>
```

```
<% # web/app/views/admin/store_products/new.html.erb %>
<div class="col-md-10 col-md-offset-1 col-sm-12 reg-page">
  <div class="headline">
    <h3><%= title('Ny produkt') %></h3>
  </div>

  <%= render('form', store_product: @store_product) %>
  <hr>
  <%= link_to('Alla produkter', admin_store_products_path, class: 'btn␣
↪secondary') %>
</div>
```

```
<% # web/app/views/admin/store_products/edit.html.erb %>
<div class="col-md-10 col-md-offset-1 col-sm-12 reg-page">
  <div class="headline">
    <h1><%= 'Redigera produkt' %></h1>
  </div>
```

(continues on next page)

(continued from previous page)

```
  <%= render('form', store_product: @store_product) %>
  <hr>
  <%= link_to('Förinta', admin_store_product_path(@store_product),
                         method: :delete,
                         data: {confirm: 'Är du säker på att du vill␣
↪förinta produkten?'},
                         class: 'btn danger pull-right') %>
  <%= link_to('Alla produkter', admin_store_products_path, class: 'btn␣
↪secondary') %>
</div>
```

7. Here, all the fields are included in the `Index` serializer.

```
# web/app/serializers/api/store_product_serializer.rb
class Api::StoreProductSerializer < ActiveModel::Serializer
  class Api::StoreProductSerializer::Index < ActiveModel::Serializer
    attributes(:id, :name, :price, :in_stock, :image_url)
  end
end
```

8. The `API controller` formats the data of each product with the implemented `StoreProductSerializer` and outputs everything as a JSON object.

```
# web/app/controllers/api/store_products_controller.rb
class Api::StoreProductsController < Api::BaseController
  load_permissions_and_authorize_resource

  def index
    @store_products = StoreProduct.all
    render json: @store_products, each_serializer:␣
↪Api::StoreProductSerializer::Index
  end
end
```

9. The ability to see all products can be done by writing:

```
# web/app/models/ability.rb
can :index, StoreProduct
```

10. Here, the API path will become `/api/store_products`. With `only` we specify that the only method we have implemented in the `API controller` is `index`.

```
# web/app/config/routes.rb
resources :store_products, only: :index
```

App

1. We create the files `app/www/store.html`, `app/www/scss/partials/_store.scss` and `app/www/js/store.js`. The JS and SCSS files are loaded by adding the respective lines.

```
<!-- app/www/index.html -->
<script type="text/javascript" src="js/store.js"></script>
```

```
// app/www/scss/index.scss
@import 'partials/store';
```

2. The route is added by specifing a `name` and `path` to the new page, as well as an `url` to the HTML file it should render.

```
// app/www/js/index.js
var alternativesView = app.views.create('#view-alternatives', {
  routesAdd: [
    // {
        // ... Other routes
    // },
    {
      name: 'store',
      path: '/store/',
      url: './store.html',
    },
    // {
        // ... Even more routes
    // }
  ]
});
```

3. Here it is important that the `data-name` is the `name` we defined in the routes, i.e. `store`.

```
<!-- app/www/store.html -->
<div data-name="store" class="page no-toolbar">
  <div class="navbar">
    <div class="navbar-inner sliding">
      <div class="left">
        <a href="#" class="back link">
          <i class="icon icon-back"></i>
          <span class="ios-only">Tillbaka</span>
        </a>
      </div>
      <div class="title">F-shoppen</div>
    </div>
  </div>
  <div class="page-content store-content">
    <div class="infinite-scroll-preloader">
      <div class="preloader"></div>
    </div>
  </div>
</div>
```

4. Here the navigation is added to the top of the alternatives view list.

```
<!-- app/www/index-html -->
<div id="view-alternatives" class="view tab">
  <div data-name="alternatives" class="page">
    <div class="navbar android-hide">
      <div class="navbar-inner sliding">
        <div class="title">Alternativ</div>
      </div>
    </div>
    <div class="page-content settings-content">
      <div class="list">
        <ul>
          <li>
            <a href="/store/" class="item-link">
              <div class="item-content">
```

(continues on next page)

(continued from previous page)

```
                <div class="item-inner">
                  <div class="item-title">F-shoppen</div>
                </div>
              </div>
            </a>
          </li>
          <!--
            ... Another list item
          //-->
        </ul>
      </div>
    </div>
  </div>
</div>
```

5. We can catch the `page:init` event when it's called on the page where `data-name="store"` by doing the following:

```
// app/www/js/store.js
$$(document).on('page:init', '.page[data-name="store"]', function () {
  console.log('Spodermon iz kewl');
});
```

6. Our JS file can now look like:

```
// app/www/js/store.js
$$(document).on('page:init', '.page[data-name="store"]', function () {
  let storeProductAPIEndpointURL = API + '/store_products';

  $.getJSON(storeProductAPIEndpointURL)
    .done(function(resp) {
      initStore(resp);
    })
    .fail(function(resp) {
      console.log(resp.statusText);
    });

  function initStore(resp) {
    console.log(resp);
  }
});
```

Here we have used the global variable `API` to define our URL. The value of `API` is defined in `app/www/js/index.js`.

7. Here we create a simple template with the `id` `storeTemplate`

```
<!-- app/www/index.html -->
<script type="text/template7" id="storeTemplate">
  Welcome to the F-store!
</script>
```

and can test if it works by extending our JS file to:

```
// app/www/js/store.js
$$(document).on('page:init', '.page[data-name="store"]', function () {
  let storeProductAPIEndpointURL = API + '/store_products';
```

(continues on next page)

(continued from previous page)

```
  $.getJSON(storeProductAPIEndpointURL)
    .done(function(resp) {
      initStore(resp);
    })
    .fail(function(resp) {
      console.log(resp.statusText);
    });

  function initStore(resp) {
    let templateHTML = app.templates.storeTemplate();
    let storeContainer = $('.store-content');
    storeContainer.html(templateHTML);
  }
});
```

Here we first get the HTML code of template and then put it into `<div class="page-content store-content"></div>` in `app/www/store.html`.

8. An example template:

```
<!-- app/www/index.html -->
<script type="text/template7" id="storeTemplate">
  {{#each products}}
    <div class="card">
      <div class="card-header" style="background-image: url({{image_url}})
↪"></div>
      <div class="card-content card-content-padding">
        <div class="product-name">{{name}}</div>
        Pris: {{price}} kr
        <button data-id="{{id}}" class="button button-fill buy-product">
↪Köp</button>
      </div>
    </div>
  {{/each}}
</script>
```

We can loop over the products and set the price to be in Swedish Kronor as:

```
// app/www/js/store.js
function initStore(resp) {
  let products = resp.store_products;
  products.forEach(function(product) {
    product.price /= 100;
    if (product.image_url === "") {
      product.image_url = "img/missing_thumb.png";
    }
  });

  let templateHTML = app.templates.storeTemplate({products: products});
  let storeContainer = $('.store-content');
  storeContainer.html(templateHTML);
}
```

Here we also set the image to be our standard missing thumbnail image if the product does not have an `image_url`.

9. Here we catch the `click` event, get the product `id` from the button and call the `buyProduct` function. The

`$(this)` is needed to get the correct `data-id`.

```
// app/www/js/store.js
function initStore(resp) {
  let products = resp.store_products;
  products.forEach(function(product) {
    product.price /= 100;
    if (product.image_url === "") {
      product.image_url = "img/missing_thumb.png";
    }
  });

  let templateHTML = app.templates.storeTemplate({products: products});
  let storeContainer = $('.store-content');
  storeContainer.html(templateHTML);

  $('.buy-product').on('click', function() {
    buyBtn = $(this);
    productId = buyBtn.attr('data-id');
    buyProduct(productId);
  });
}

function buyProduct(id) {
  $.ajax({
    url: API + '/store_orders',
    type: 'POST',
    dataType: 'json',
    data: {
      "item": {
        "id": id,
        "quantity": 1
      }
    },
    success: function(resp) {
      app.dialog.alert(resp.success, 'Varan är köpt');
    },
    error: function(resp) {
      app.dialog.alert(resp.responseJSON.error);
    }
  });
}
```

10. SCSS code and the complete JS file:

```
// app/www/scss/partials/_store.scss
.store-content {
  .card:nth-child(-n+2) {
    margin-top: 16px;
  }

  .card {
    width: calc(50% - 18px);
    float: left;
    box-shadow: none;
    margin-left: 8px
  }

```

(continues on next page)

(continued from previous page)

```
  .card-header {
    background-size: cover;
    background-repeat: no-repeat;
    background-position: center;
    background-color: #f8f8f8;
    height: 37vh;
  }

  .card-content {
    text-align: center;
  }

  .product-name {
    font-size: 19px;
    font-weight: bold;
  }

  .buy-product {
    background-color: $fsek-orange;
    margin-top: 10px;
  }
}
```

```
// app/www/js/store.js
$$(document).on('page:init', '.page[data-name="store"]', function () {
  let storeProductAPIEndpointURL = API + '/store_products';

  $.getJSON(storeProductAPIEndpointURL)
    .done(function(resp) {
      initStore(resp);
    })
    .fail(function(resp) {
      console.log(resp.statusText);
    });

  function initStore(resp) {
    let products = resp.store_products;
    products.forEach(function(product) {
      product.price /= 100;
      if (product.image_url === "") {
        product.image_url = "img/missing_thumb.png";
      }
    });

    let templateHTML = app.templates.storeTemplate({products: products});
    let storeContainer = $('.store-content');
    storeContainer.html(templateHTML);

    $('.buy-product').on('click', function() {
      buyBtn = $(this);
      productId = buyBtn.attr('data-id');
      buyProduct(productId);
    });
  }

  function buyProduct(id) {
```

(continues on next page)

(continued from previous page)

```
    $.ajax({
      url: API + '/store_orders',
      type: 'POST',
      dataType: 'json',
      data: {
        "item": {
          "id": id,
          "quantity": 1
        }
      },
      success: function(resp) {
        app.dialog.alert(resp.success, 'Varan är köpt');
      },
      error: function(resp) {
        app.dialog.alert(resp.responseJSON.error);
      }
    });
  }
});
```

7.2 Spider Conference 2018

The Spider Conference 2018 was the first Spider Conference. The task was to implement a quiz. Necessary preparations should be done in the backend to fetch new questions and display them in the app. The idea is then that when the user selects an alternative to the question, a request is sent from the app to the server which will respond if it was correct or not. If the alternative was wrong, then it should send an error and the user has to restart from question one.

Here one can find a detailed description of the task. The solution to this task is left as an exercise for the interested Spiderman.

CHAPTER 8

How to be a spodermon

Spodermons do other things than just program awesome applications. We also answer web related emails, handle mail alias, approve memberships etc. This is all described in the Things spodermons do page (in Swedish).

8.1 Godkänna användare

Då medlemmar hör av sig till spindelmännen om att de inte har behörighet någonstans eller att de inte kan använda någon funktion på hemsidan är det troligtvis så att de inte är godkända användare. Detta är enkelt löst.

Steg 1: Gå in på ‘’Administrera”-menyn och välj ‘’Användare”.

(BILD)

Steg 2: Klicka ‘’Ge medlemskap”

(BILD)

Klar!

8.2 Create mail alias

There is a lot of members that what ot have their own cool mail alias for their blabla in the F-guild. Note that this is only an alias and **not** an new email. These aliases only forward mail to an already excisting email adress. To create a new mail alias you use the website. We can create aliases for name@fsektionen.se or name@farad.nu.

Step 1: Log into the website and go to “Administrate” menu and choose “Mail aliases” in the “User” column.

Det är många medlemmar som vill ha egna coola mailalias för sitt engagemang på F-sektionen. Observera dock att detta endast är ett alias och INTE en ny mailadress. Dessa mailalias bara vidarebefodrar till en existerande emailadress som medlemmen äger. Skapa mailalias gör du via hemsidan. Vi kan skapa mailalias för blablabla@fsektionen.se eller blablabla@farad.nu.

Steg 1: Gå in på ‘’Administrera”-menyn och välj ‘’Mailalias”.

(BILD)

Steg 2: Skriv in önskad mailalias i sökrutan och klicka sök.

Steg 3: Skriv in mailadress som önskas kopplas till mailaliaset. Glöm inte att trycka på “Spara”-knappen!

(BILD)

Klar!

8.3 Ändra mailadresser kopplade till mailalias

Steg 1: Gå in på ‘’Administrera”-menyn och välj ‘’Mailalias”.

(BILD)

Steg 2: Klicka på “Redigera”-knappen så öppnas ett fält med nuvarande mailadresser kopplade till mailaliaset.

(BILD)

Steg 3: Skriv till en mailadress på en ny rad eller ändra en befintlig. Glöm inte att trycka på “Spara”-ikonen!

Klar!

8.4 Skapa snabblänk

Steg 1: Gå in på ‘’Administrera”-menyn och välj ‘’Snabblänkar”.

(BILD)

Steg 2: Skriv in namn på snabblänken i vänster fält och vart snabblänken ska leda i höger fält. Alltså, snabblänken kommer se ut fsektionen.se/google och när man klickar på denna så kommer man till www.google.com

(BILD)

Klar!

8.5 Svara på mail

Steg 1: Läs mailet.

Steg 2: Tänk ut ett bra svar på mailet som är hjälpsamt.

Steg 3: Skriv detta bra-iga svar och tänk på att vidarebefodra svaret till alla spindelmän!

Steg 4: Du har nu +1000 poäng i Jakobs bok.

Klar!

8.6 Gå på möte

Steg 1: Lägg in i din kalender vilka dagar som det är spindelmöte. Du kan också följa spindelkalendern.

Steg 2: Kom till spindelmötet. Var inte sen!

Steg 3: Ät fika. Mums!

Steg 4: Gör spindelarbete.

Klar!

CHAPTER 9

How to prepare the Rostsystem

First and foremost you must install Heroku CLI. Once installed run `heroku login` and login using the Spider foreman account.

1. Run cleanup task using the Heroku CLI with `heroku run rake cleanup:keep_users --app fsekvoting`
2. Change dyno type from “Free” to “Hobby”. This should be done a day or a couple of hours before the assembly, and then switched back to “Free” when it is over since the “Hobby” dyno costs money
3. If needed, give rights to the guild’s secretary on the website so they can set the correct roles to people attending the assembly
4. (Optional) The database can store 10000 rows, but if this is not enough then you have to upgrade the Postgres add-on from “Hobby Dev” to “Hobby Basic”. 10000 rows is however often more than enough for a guild assembly. The number of rows used can be seen by typing `heroku pg:info --app fsekvoting`

CHAPTER 10

This is a sick title

This is some text with **a bold word** and *a word in italic*.

This is a link to a cool place.

This is a *link* to an another page in the docs through a reference label defined in the top of the getting_started.rst file.

10.1 This is an ordinary header

Lorem ipsum blablabla.

10.1.1 This subheader is quite nice

Now I would like to insert some code for the user :) :

```
[{firstLetter: A, songs: [song1, song2, ...]}, {firstLetter: B, ..}, ...]

sphinx-build -b html -a ./source ./build
```

and this is some other code:

```
var listIndex = app.listIndex.create({
  el: '.list-index',
  listEl: '#songbook-list',
  label: true,
});
```

and here is some `super cool inline code highlights`.

10.1.2 Maybe you want a second subheader

Make sure that this .rst file is linked in app or web. See the documentation for reStructured for more things.

1. This is a numbered list.
2. It has two items.

- This is a bulleted list.
- It has two items too, the second item uses two lines.

1. This is an another numbered list.
2. It has two items too.

and here comes a table :O

Header row, column 1 (header rows optional)	Header 2	Header 3	Header 4
body row 1, column 1	column 2	column 3	column 4
body row 2	...	...	

10.2 Local API

If you have configured the website you can connect to the local web server instead of the one running stage and thus develop and test an api directly by changing stage.fsektionen,se/api to tcp:3000/api in index.js:

```
// API URLS
const BASE_URL = 'tcp://localhost:3000'
```

10.3 Phonegap

To get PhoneGap set up we can start with PhoneGap's own installation guide. Do step 1 in this guide and install the PhoneGap CLI (Command Line Interface) and not the desktop application. Be sure to install the requirements stated in the guide (Node.js and Git) before you install the actual CLI. The reason why we don't use the desktop app due to its limitations and the same goes for Github's desktop application. It's important to note that it takes a while to get used to these CLI:s, Github especially, because they do not have an intuitive interface. Rather, they have a lot of different commands that take some time to learn but are better in the long run. With that in mind don't be afraid to ask for help if things ever become confusing. Step 2 of the guide is installing PhoneGap's mobile app, which is a way for you to test your changes on your own device. We haven't tested it to a large extent yet, but it's basically the equivalent of a simple emulator that is very easy to set up. For now, this will do perfectly fine though and I would recommend downloading it. How to use their app is described later on in Standard workflow.

10.4 Cloning the Github repository

A repository or repo is, in other words, a Github project where all of our files are stored. When we talk about cloning a repo we mean downloading the project files to your computer and linking them with Github. We do this by using the Github command clone that is executed in CMD on Windows or the Terminal for macOS and Linux based systems. When executing a Github command you always use the prefix git i.e. the syntax would be git [command]. Now, when cloning we also need to specify what repo we're going to clone. We can do this by using a special link that you can find on the repo's start page (the code tab), which is here for the F-app. On the right, there should be a big green button saying "Clone or download" that will give you the cloning link (see figure below). Make sure it says Clone with HTTPS at the top and not Clone with SSH.

Cloning repository (Picture)

The last thing you need to do is to pick what directory you're cloning into i.e. where you want to save the project files. Personally, I have a folder in documents with both the app and web repo in it called "Spindel". I will show you how you can do this as well, but if you have another preference that's fine too. For example, you might want to save into the home folder instead of documents on Linux.

Open the CMD/Terminal and navigate to the directory of your choice. The commands that will be used to demonstrate this are dir, ls, cd and mkdir. On the left-hand side of the CMD/Terminal the current directory is displayed. The commands dir(Windows) and ls (Mac/Linux) does the same thing for different OS and shows the current directory's content. cd is a command that changes the current directory with the syntax cd [directory]. For example, if your current directory is C:Usersfredrik then dir would show a list of all files and folders in your fredrik user folder, one of which would be Documents. If you want to enter Documents, i.e. change your current directory to Documents, you would execute:

```
cd Documents
```

and the left-hand side would change to C:UsersfredrDocuments. You can always go up one level in the directory tree with cd .. and, in this case, return to C:Usersfredr. While in Documents you can execute:

```
mkdir Spindel
```

and create a new folder (make a directory) called "Spindel" in Documents. dir should now also show Spindel and cd Spindel will change the current directory to C:UsersfredrDocumentsSpindel. Note that this will look a bit different for Mac and Linux and C:Usersfredrwill be replaced with ~. Now, when you've navigated to the directory you want to clone into, execute:

```
git clone [repo cloning link]
```

and you should see it start downloading files. Note, that you should not keep the parentheses and replace [repo cloning link] with the link. Feel free to start with the next step while you're downloading the files. If you've done everything correctly you should have a new app folder in the directory you chose.

10.5 Creating a custom CMD/Terminal command

When you've successfully cloned the repo you'll have a local app folder somewhere on your computer containing all the project files. Every time you start working you need to access that specific directory through the CMD/Terminal to host a web server (this will be explained further in Standard workflow). Therefore, it's a good idea to make a custom command that will take you to this directory directly, so you'll avoid navigating there manually every time. How to do this is explained below for different operating systems. Note that this step is optional if you've cloned into, or close to, the home directory and don't mind navigating to the project every time.

First, open the project folder in the file explorer, navigate to the www folder and copy its address from the top. The address should look something like C:. . . appwww. Open notepad, type cd and paste the address. The file should now be looking something like:

```
cd C:\...\app\www
```

Save the file on your desktop as apploc.cmd or with whatever name you see fit. Note that the filename must end with .cmd and have the type All Files (see figure below).

Saving custom command file windows (picture)

Now, go back to the file explorer and navigate to your nodejs folder in Program/Program Files. It should have an address close to C:Program Filesnodejs. Move the apploc.cmd file into that folder and try executing it in CMD by typing apploc and hitting return. If everything works you should see the current directory on the left-hand side change to the project's www folder.

On Mac and Linux you'll do everything through the terminal, so open that up and make sure you're in your home directory. ~ is the notation for the home directory and should be shown as the current directory in the command prompt. Normally you're already in your home directory when you start the terminal, but if not execute the following to go there:

```
cd ~
```

Then, use ls -a to list all the files in the home directory. Adding the flag -a will show all the files, even the hidden ones which are marked with a dot prefix. After you've run ls you should see a file called .bash_profile on Mac or .bashrc on Linux. If you don't see it you need to create it. You do this with the command touch as shown below:

```
// Mac
touch .bash_profile

// Linux
touch .bashrc
```

Before we continue with the bash file you need to find the www folder in the app folder you've just cloned and copy its directory path. The easiest way to do this is to find it in Files, enter the www folder, right-click any file or folder inside it, go to Properties and copy the location string. Now, to create a new command you need to define it in the bash file. The best way to do this is by a terminal text editor, like Nano or Vim. You open the file with Nano by entering:

```
// Mac
nano .bash_profile

// Linux
nano .bashrc
```

Now, navigate to the bottom of the file (it might be empty) with your arrow keys and add:

```
alias apploc='cd [www folder path]'
```

where [www folder path] is the location string you've just copied. To save the file you first exit by pressing Ctrl-X on your keyboard. It will then ask you if you want to save where you press your y key for yes. Lastly, it will ask you what you want to save it as, which should be .bashrc and already filled in. Then, press enter to save the file. Restart the terminal and try executing apploc. If you've done everything correctly your current directory should now be the www folder.