EcoBeach Group Part

Semesterproject in Scalable Systems (E21)

Gábor Gulásci
gagul21

Niels Faurskov
niean15

Nikolai Emil Damm
nidam16

Zsófia Bardócz
zsbar21

January 3, 2022

Character count with spaces: 25230 ≈ 10,5 pages.

Contents

Chapter 1: The Problem

In this chapter, we will describe the problem addressed by EcoBeach, a system built as part of the Semester Project in Scalable Systems, on the first semester on the masters of Software Engineering SDU.

First, a problem definition will be given, along with the overall objective. Next, an in-depth problem description is presented, where the sub-problems will be unveiled, and why it is necessary to derive a solution.

1.1 Problem and Objective

At present, the ecology is threatened by increasing changes to the climate. One of the notable climate changes is the rising shorelines that are predicted to rise to critical heights in the 21st century. According to the publication, Sea level rise and its coastal impacts, by Anny Cazenave and Gonéri Le Cozannet, we will see an average increase of 40-75 cm on a global scale by the year 2100 [1, p. 23]. This increase will not happen uniformly, and some areas will see higher sea levels than others [1, p. 21], which can result in sea level increases of up 30% in some regions [1, p. 23]. In Figure 1.1 a projection of the expected sea-level rise by the year 2100 is illustrated.

As rising shorelines are a growing threat, it is imperative to react and minimize climate change, but sadly, this might not be something humanity can do on time. Therefore, monitoring how the shorelines are changing is paramount to alleviate the risk of rising shorelines. This growing threat and the accompanying concerns preface the problem that is:

> Humanity might not combat climate change to avoid massive floods in cities and countries. Currently, there are not enough accessible options to monitor how shorelines are changing to prepare for or predict floods.

This project aims to solve this problem by creating a system capable of processing satellite imagery of geo-locations worldwide in real-time to determine how the shorelines have changed

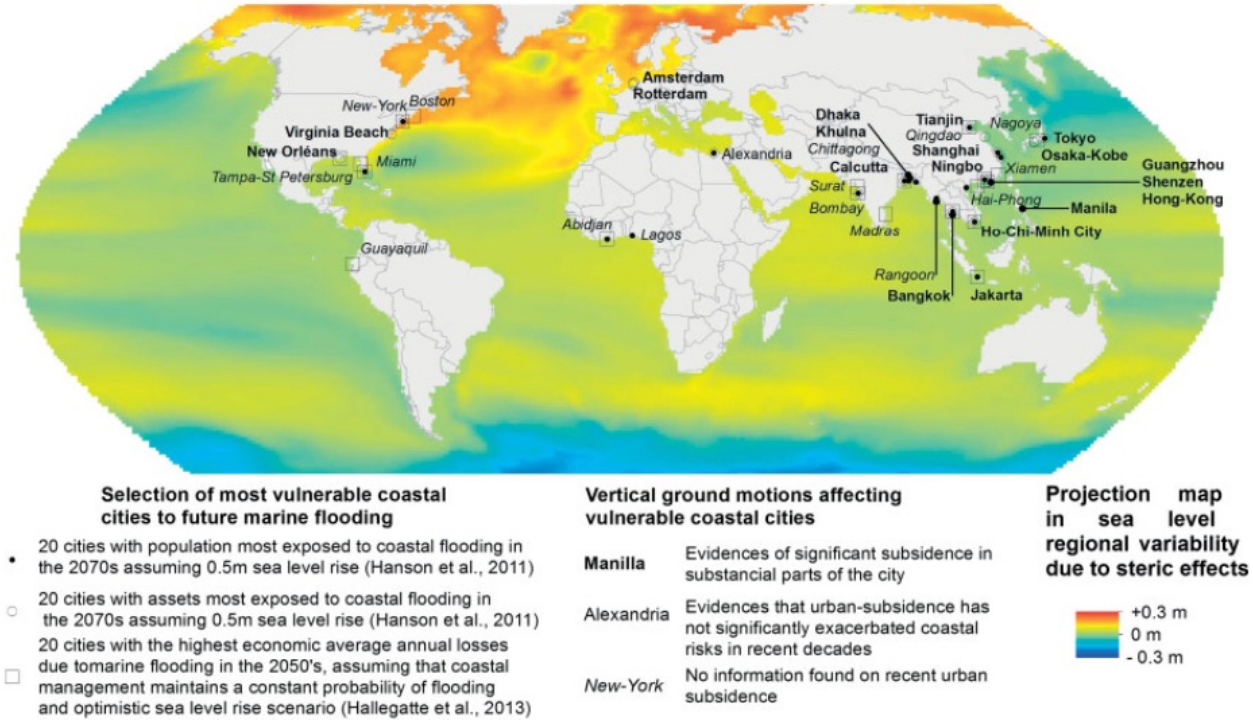

Figure 1.1: A projection of expected sea level regional variability by year 2100. Taken from, Sea level rise and its coastal impacts, by Anny Cazenave and Gonéri Le Cozannet [1, p. 27] (Copyrightable under the terms of the Creative Commons Attribution-NonCommercial-NoDerivs License)

and are changing.

The project will rely on mobile sensing and various big data technologies and tools to create such a system.

1.2 Problem Description

Monitoring shorelines is quite attractive; having historical and current data on shoreline changes can potentially help both the public and the private sector.

In the 21st century, governments will need to develop new and ingenious ways to combat the rising shorelines and avoid floods. Knowing where shorelines are rising the most can be a huge factor in helping decide where to preemptively build solutions that can help prevent large floodings and the destruction of properties and, potentially, cities. Some governments already have needed to find solutions to flooding, which is apparent when looking at the Netherlands that have built resilient solutions to prevent floods, e.g., the Maeslantkering

storm surge barrier [2].

It might also be beneficial to know how shorelines are changing in the private sector. Knowing this can be helpful to make informed decisions about where to settle down or prepare for floods for individuals living at places at risk.

From a technical perspective, creating a system capable of processing satellite imagery in real-time is a daunting task with many sub-problems that need to be solved to make it feasible. To derive a solution, we believe the following sub-problems must be solved:

- How to download satellite images from Copernicus?
- How to process images, so water is differentiated from land?
- How to build a system capable of handling big data?
- How to build a system capable of real-time processing?
- How to build a highly resilient system?
- How to build a mobile application that uses mobile sensing in a meaningful way to visualize shoreline changes?

It is paramount that these problems can be solved to derive a feasible solution to monitor shorelines in real-time. Such a solution, EcoBeach, is described in detail in the next chapter.

Chapter 2: The Solution

This chapter will present EcoBeach, a highly resilient distributed system that can process satellite imagery in real-time and analyze the difference in water levels on given geo-locations. EcoBeach collects data from beach locations (hence the name EcoBeach). Limiting geo locations to beaches in select countries is a deliberate choice, as the available server resources currently restrict the solution's scalability.

In section 2.1 Solution Approach EcoBeach will be described at a high-level. First, a summary of solutions to the sub-problems in section 1.2 is given. Next, the infrastructure of EcoBeach is presented to show how the solutions to each of the sub-problems complement each other to provide a feasible solution to monitoring shorelines.

Lastly, the advantages and disadvantages of EcoBeach will be presented.

2.1 Solution Approach

EcoBeach consists of services and applications that play a crucial role in monitoring shorelines. There is a total of 11 services/applications as shown in Table 2.1.
The services in EcoBeach were chosen or created to provide solutions to the sub-problems. Below a summary of each solution to the sub-problems is presented.

How to download satellite images from Copernicus? To download satellite images, we created a resilient scraping service that continuously downloads and pre-processes satellite images on given geo-locations. The scraping service is described in detail in subsection 2.2.3.

How to process images so water is differentiated from land? For this purpose, we pre-process images during the scraping process, so they are saved as black and white images according to their Normalized Difference Water Index (NDWI) value. Then we created the Spark NDWI Analyzer, a Spark Job, that analyzes downloaded satellite images to determine what is water or land. The Spark NDWI Analyzer is described in subsection 2.2.4.

Service name	**Service description**
Sentinel Satellite Scraper	A python script that downloads and pre-processes satellite imagery
Spark NDWI Analyzer	A Spark job that analyzes pre-processed satellite images for shoreline changes.
Kafka	A distributed event streaming service, where intermediary data is saved as part of the processing pipeline.
Spark	A large-scale data analytics framework that supports publishing jobs that are processed on distributed Spark Workers.
Hadoop	A framework that allows distributed file storage primarily with HDFS. Used for saving checkpoints and the pre-processed satellite images.
Zookeeper	A centralized service to enable reliable distributed coordination for Hadoop and Kafka.
MongoDB	A distributed database to save and query fully processed data.
MongoDB Kafka Connector	A sink connector for Kafka to feed fully processed data from Kafka to MongoDB
Kowl	An intuitive monitoring service that allows viewing and configuring running Kafka services.
WebAPI	A .NET WebAPI to provide a nice interface for querying data from MongoDB
Android Application	The EcoBeach app where fully processed data is represented with the google maps interface.

Table 2.1: The services/applications in the EcoBeach system

How to build a system capable of handling big data? To create a system capable of large-scale data processing and analysis, we created a stack that relies on distributed systems that are very scalable and fault-tolerant. The stack includes Kafka, Spark, Hadoop, Zookeeper, and MongoDB and is described in subsection 2.2.1

How to build a system capable of real-time processing? The main contributor to this is Kafka and Spark. Kafka allows us to create topics with intermediary data, where our Spark NDWI Analyzer Spark Job is set up as a consumer that processes new entries as they are created. Together with the rest of the stack, it allows us to create, process, and feed data to MongoDB quickly and reliably as new data is entering the system.

How to build a highly resilient system? To make EcoBeach a resilient system, we identified single-points of failure and added load-balancing and distribution of services to ensure that the system would function reliably in case of failures. Docker Swarm as the chosen container orchestration tool was a massive help in configuring this. This is described in more detail in subsection 2.2.2.

How to build a mobile application that uses mobile sensing in a meaningful way to visualize shoreline changes? To make the EcoBeach app utilize mobile sensing, we made the app rely on the user's current location and provide data accordingly. As the EcoBeach app is an Android application, the Google Maps API is what the app relies on to provide most of its features. The EcoBeach app is described in detail in subsection 2.2.6

2.1.1 Advantages and Disadvantages

The applications, the cluster, and the features they provide make for a robust configuration that is scalable. Our infrastructure can easily handle the limited intake retrieved from the Sentinel 2 API. Additionally, the current setup incorporates redundancy in many levels; as such, the database, event processing (Kafka), and the scrapers can easily recover if a service should fail; this will be further explored within subsection 2.2.2.
The android app provides a clear view of the available data points and gives the user instant access to relevant metrics surrounding the beaches. Additionally, the setup can download and examine historical data, which allows for analytics. These analytics can, in the future, be used to track and extrapolate the changes occurring over time, which could, in turn, be used to estimate the future of beach health.

The disadvantage of this approach is that the scrapers can often sit and wait for incoming data. The satellite imagery is not real-time, and thus the solution cannot offer real-time user alerts. The android app currently shows limited information gathered and should contain more contextual information for the end-user.

2.2 Solution Description

In this section, an in-depth description of the solution is presented. First, the hosting setup and the central services that drive the data pipeline are explained. After this, the services that create, read, process, and visualize data in EcoBeach are presented.

2.2.1 Big Data stack

When planning our project, distributed data storage and redundancy was one of the most important aspects we were looking to implement. Our cluster runs on multiple servers - or nodes - with one designated as master - or name node - to provide redundancy and reduce, or completely eliminite risk of data loss. Since the sheer size of the data that we request and analyze from the Sentinel2 API demands parallel processing, we also implemented the necessary tools to provide a seamless streaming of data.

HDFS

When image scrapers are running on all the servers, they store their data in the Hadoop Distributed File System. Our HDFS consists of three servers, out of which our Helsinki server is the name node, and all three servers are data nodes. The name node manages the filesystem meta-data, while the three data nodes store the actual data.

Kafka

Kafka is running on all servers and connects individual parts of our clusters via Kafka Connect streams. We used Kafka topics to differentiate between the origin of the data, whether it is the raw data coming from the NDWI scrapers or the analyzed images. Kafka provides us with the opportunity to stream our data within the cluster.

Spark

We use Apache Spark for our image analyzers which helps us process the data in batches via a continuous process stream. The data is consumed from Kafka continously and distributed

Spark workers run a python job for image analyzation.

MongoDb

Our choice of distributed database is MongoDB with one replica set that replicates data on Helsinki and Nuremberg. MongoDB is running on two server instances for extra redundancy. As a document-based NoSQL database, it stores its objects in JSON format, making it easier to communicate data to the API.

2.2.2 The infrastructure

This section describes the solution's infrastructure, how everything is tied together, how services communicate, and this setup's resilience. Additionally, it aims to give the reader a better understanding of the need for the different services and sheds light on why the specific setup was selected.

The infrastructure of EcoBeach consists of three servers that are placed in different cities within the EU, hereunder:

- Helsinki, Finland
- Falkenstein, Germany
- Nuremberg, Germany

These servers make up the EcoBeach cluster and will be referenced as the cluster henceforth. Each server is unmanaged and runs a clean Ubuntu 20.04 installation supplied by Hetzner. At the time of setup, Hetzner's pool of servers was only available within Germany and Finland, hence the three servers' placement. The servers are placed within a subnet, allowing them to communicate securely and without interference. Nodes and servers will be used interchangeably throughout the remainder of this section.

For a more detailed overview of the setup, please consult the infrastructure diagram in Appendix A, which visualizes the interactions occurring between the services and how they relate to each other.

Container Orchestration

Setting up services across many nodes can be cumbersome and painful; additionally, configuring each service manually increases the risk of incorrectly setting up the configuration. A bad configuration can easily result in many hours spent debugging or the cluster becoming

unstable down the road until the root cause is found. Today, many of these pain points can be alleviated by container orchestration, which takes care of the communication between the nodes, and enables a master node to distribute services across the cluster intelligently.

The EcoBeach cluster uses Docker Swarm, mainly because of the group's current knowledge and experience with docker. Also, its interoperability with docker-compose files makes orchestration a breeze with most services. Docker Swarm also takes care of load-balancing - once a service is exposed externally, docker will automatically make sure incoming requests are distributed equally by round-robin selection. Additionally, Docker Swarm is fault-tolerant and will automatically bring up a service that may have halted unless otherwise specified.

Docker Swarm also allows for placement constraints, which is incredibly useful when a specific service should only be available on a specific node.

Deployment of the project's codebase is relatively easy with docker swarm. Code that should be deployed is wrapped in a docker image and then uploaded to a docker registry. Afterward, the image can be used as a distributed service within the cluster by including it as a docker-compose file.

The entirety of EcoBeach's cluster, except our database, is being managed by Docker Swarm.

Management of services in the cluster

The following services are managed through docker swarm:

- Kafka (and ZooKeeper)
- Apache Hadoop HDFS
- Apache Spark
- EcoBeach Codebase
  - NDWI Scraper
  - Image Analyzer
  - Rest API

Besides these services, our MongoDB instance is manually managed due to intricate settings.

Data ingestion and travel path

The data ingestion and the application's entry point occur within the NDWI scraper. The NDWI scraper services are never exposed externally; however, they communicate internally with our Kafka services and regularly publish to specific Kafka topics.

The scrapers continually download satellite imagery, processes it into grayscale images, and publish it to Kafka for further operations. For a more in-depth description of how the Sentinel Satellite Scraper (SSS) works see subsection 2.2.3.

While these services are not necessarily scraping all the time, the services are always up and running, making them highly available. The services are also replicated across the cluster since retrieving the relevant data is often a slow process with the sentinel 2 API. When replicated, the data intake and throughput are increased. This increase makes much sense since the cluster can handle a much larger intake than the amount of relevant data provided by the Sentinel 2 API. Additionally, this setup has the added benefit of also allowing for more granular control, as each replicated service is responsible for a different region. If the cluster's performance is under strain, a region can be disabled temporarily and reenabled at another time when the cluster is no longer under load.

Event streaming

When the number of services within the cluster grows, intercommunication can become troublesome and, at worst, force one to continually make cumbersome changes to each service until communication is made. The solution to this problem is Kafka. Kafka follows the pub-sub event pattern and thus provides a stream with N amounts of topics that the services can publish or subscribe to.

The cluster runs three Kafka brokers, one on each server. Additionally, the Helsinki node runs zookeeper, which (amongst many other things) synchronizes incoming jobs and keeps track of what is Kafka is doing across the cluster. Having brokers on each node introduces redundancy and allows for better distribution of the incoming/outgoing messages. Kafka also makes the cluster very scalable, as newer services can start consuming on different topics if need be, without necessarily interfering with current working services. Another benefit to Kafka is Kafka Connect, which integrates external services by installing a specific plugin. Once set up, sources and sinks can be created that Kafka can produce to or consume from respectively.

Communication between the brokers and the other services is currently limited to internal networking. Not only does this make the infrastructure more secure, but since all the services reside on the same network, there is simply no need for it to be publicly accessible.

The Kafka brokers are used by:

- The NDWI Scraper (Producer):
  - Produces whenever an image matches the filter, metadata and grayscale are produced to the "ndwi_images" topic.
- The Image Analyzer (Producer and Consumer):
  - Consumes on the "ndwi_images" topic; thus, whenever an image is available, this service will analyze the image to better understand the amount of water to shore ratio within the area supplied in the image.
  - Produces the resulting data set onto the "ndwi_results" topic.
- MongoDB connector (Consumer):
  - Consumes items from the "ndwi_results" topic and copies the contents to the MongoDB database through Kafka Connect.

Data storage

The database of choice was settled on MongoDB. MongoDB is great in a big data context, as it scales incredibly well, with built-in tooling for redundancy and sharding, which is often lacking in many other DBMS solutions, or left to external plugins with added overhead. Furthermore, MongoDB being document-oriented makes it straightforward to store data of different structures, as in the case of the data coming from our Kafka Connect connector. MongoDB is also well supported within the developer community, making it possible to connect and communicate with the database in many different languages and frameworks.

MongoDB is the only service that is manually managed due to some of the intricate configurations that had to be done to connect the different nodes. Like the other services, the database is also only internally available within the cluster to enhance security.

The MongoDB database is placed on the Helsinki and Falkenstein nodes, configured as a replica set. MongoDB will automatically replicate data across the two databases, and should one fail, the other can take over to maximize uptime and make the entire setup fault-safe.

External availability

The only services that are externally available is the rest API. The rest API retrieves data from our database and exposes it publicly. Front-end applications can then consume the API, and in this case, our Android app relies on the information given in the API.

2.2.3 Sentinel Satellite Scraper

The Sentinel Satellite Scraper (SSS) is a python script created to scrape and continuously download satellite imagery from given geo-locations. The script was created as EcoBeach relies on the analysis of geo-locations to determine how shorelines are changing. Because of this, it requires a steady input of data to reliable show how shorelines historically and currently have changed.

The SSS runs every week to scrape satellite images for beaches in Denmark, Sweden, Germany, and Great Britain. Each subsequent run scrapes imagery three years back from the current date. It only downloads images that the EcoBeach pipeline has not previously processed by caching completed work. Subsequent runs are much faster due to this caching strategy.

To understand the intricacies of the SSS, one must understand the format of the data downloaded from the Sentinel Satellite. It is essential to mention the script downloads satellite imagery from the Sentinel-2 satellite.

Sentinel-2 satellite data products

The Sentinel-2 satellite has two types of products available for download. The two product types are Level-1C and Level-2A as illustrated in Table 2.2.

Product name	**High-level Description**	**Data Volume**
Level-1C	Top-of-atmosphere	600MB — 100x100 km2
Level-2A	Bottom-of-atmosphere	800MB — 100x100 km2

Table 2.2: The different products available from the Sentinel-2 sattelite. Adapted from Sentinel-2 data products

EcoBeach relies on Level-2A products, as these are ground images and not atmospheric images. The Level-2A products can be downloaded in 3 different spatial resolutions, 10m, 20m, 60m, all as tiles covering 100x100 km^2. The spatial resolution defines how many cubic meters each pixel covers and what spectral bands are available. Naturally, the lower the spatial resolution is, the more detailed the product is. However, it also increases its size to $\sim 1GB$ per product at 10m spatial resolution.

As previously mentioned, each spatial resolution contains different spectral bands. The 10m spectral resolution is the one the SSS uses, and it includes four spectral bands: band 2, band 3, band 4, and band 8, which defines the different wavelengths the Sentinel-2 Satellite can capture. The specifications of these bands can be seen in Table 2.3. [3]

Band	**Central wavelength**	**Color**
B2	496.6 nm	Blue
B3	560.0 nm	Green
B4	664.5 nm	Red
B8	836.1 nm	Visible and Near Infrared (VNIR)

Table 2.3: The four spectral bands included with the 10m spectral resolution. Adapted from Sentinel-2 data products

The implementation and design

SSS uses quite a few python libraries to download products, combine bands, pre-processing, and produce Kafka messages. The first and arguably the most important is the *sentinelloader* library. The *sentinelloader* library is in control of the logic related to downloading the products from Sentinel-2, combining bands, and cropping the resulting image to the location of interest. Lets first have a look at the entry point of the script the *scrape(args)* method in Figure 2.1.

The scraping method first instantiates the *Sentinel2Loader* from the *sentinelloader* library with the folder to download products too, the Copernicus user, the desired cloud coverage percentage, and the log level. After initialisation the locations to scrape for are read in from one of many excel files located in the scrapers directory. Each of these Excel files contains geo-locations on beaches in a specific country and come from a publicly available dataset from the European Environmental Agency [4].

When the Excel data has been read to memory, the main scraping loop begins, where it randomly selects a non-processed beach and proceeds. Randomly selecting a beach is a deliberate choice as the Copernicus API restricts the number of offline products that can be downloaded to 20 each day. Mixing up which locations we query historical data from ensures we use the offline product retrievals differently each time the script is run. Thus, we improve the chances of getting historical data for different areas over time.

Next, the different needed parameters are extracted from the excel data, and the process of downloading products, combining bands, and cropping the resulting images is started by

```python
def scrape(args):
    sl = sl2.Sentinel2Loader('downloads', user, passw,
                             cloudCoverage=(0, 1), loglevel=logging.INFO)
    dfs = pd.read_excel(
        f"beach_datasets/{args.countrycode}.xlsx", sheet_name=args.countrycode)

    for ri in randomIndexesInDfs(dfs):
        countryCode = dfs.iloc[ri][0]
        locationName = dfs.iloc[ri][3].replace(".", "").replace("/", "_").title()
        lon = dfs.iloc[ri][6]
        lat = dfs.iloc[ri][7]

        today = date.today()
        geoTiffImages = sl.getRegionHistory(
                countryCode,
                locationName,
                createSearchArea(lon, lat, 2),
                'NDWI_MacFeeters',
                '10m',
                str(today - timedelta(days=args.days)),
                str(today)
            )
        processGeoTiffImages(
            args, countryCode, locationName, lon, lat, geoTiffImages)
        removeDownloadFolder()
```

Figure 2.1: The *scrape(args)* methodh that is the entry point of the SSS.

the *getRegionHistory(...)* method call. This call is called with the country code, the location name, a search area that defines the boundary of the location of interest, the index we want to combine bands, the spatial resolution, the from date, and lastly, the to date.

As indicated by the "NDWI_MacFeeters" parameter, the combination of bands happens accordingly to the Normalized Difference Water Index (NDWI) [5]. The MacFeeters formula for NDWI combines the bands into an image that differentiates water from land. The NDWI uses a simple formula to combine band 3 and band 8 to give a value between 0 and 1 for each pixel that indicates how likely that pixel is to be either water or land. [6]

$$NDWI = \frac{B3 - B8}{B3 + B8}$$

Furthermore, the cropping of the image relies on the provided search area that defines the boundaries of the location of interest. *Sentinelloader* automatically crops out the area of

interest from the downloaded product(s). In cases where the place of interest covers multiple products, the *sentinelloader* library can download, combine and crop the image from various products, so no data is lost.

After having downloaded products, combined bands, and cropped the resulting image, custom pre-processing is done on the resulting image to color it based on the NDWI values and to ensure the image size is as compact as possible. This work happens in the *processGeoTiffImages(...)*.

```
def processGeoTiffImages(args, countryCode, locationName, lon, lat, geoTiffImages):
    for geoTiffImage in geoTiffImages:
        # gets the date part from the geoTiff path
        date = geoTiffImage.split("-NDWI_MacFeeters")[0].split("tmp/")[1]
        imageName = f"{countryCode}-{locationName}-{date}.png"
        imagePath = f"processed/{imageName}"
        createProcessingFolder()
        processImage(args, countryCode, locationName, lon, lat,
                     geoTiffImage, date, imageName, imagePath)
        removeProcessedImage(geoTiffImage)

def processImage(args, countryCode, locationName, lon, lat, geoTiff, geoTiffDate, imageName, imagePath):
    # We only want create and publish new images.
    if(not processedImageExists(imagePath)):
        createBlackAndWhiteImg(geoTiff, imagePath)
        publishToKafkaTopic(args.kafka_servers, countryCode, locationName, [
                            lon, lat], geoTiffDate, imageName)
```

Figure 2.2: The *scrape(args)* method that is the entry point of the SSS.

First, the desired image name and path of the pre-processed images are constructed from the downloaded images from the *sentinelloader*. Then the downloaded images are passed on to the *processImage(...)* method that first creates a new image from one of the downloaded images. The new image is combined with a black and white color map that paints any pixel with an NDWI value greater than 0.6 white, and any pixel with an NDWI value smaller than 0.6 black. Two of these images can be seen in Figure 2.3.

Finally, the image and relevant metadata are compressed to JSON and published to Kafka. After this, all downloaded satellite imagery is deleted to clear up valuable space for processing the following location. If this cleanup process is skipped, the downloaded products would quickly fill up the remaining disk space on the machines where the script is run.

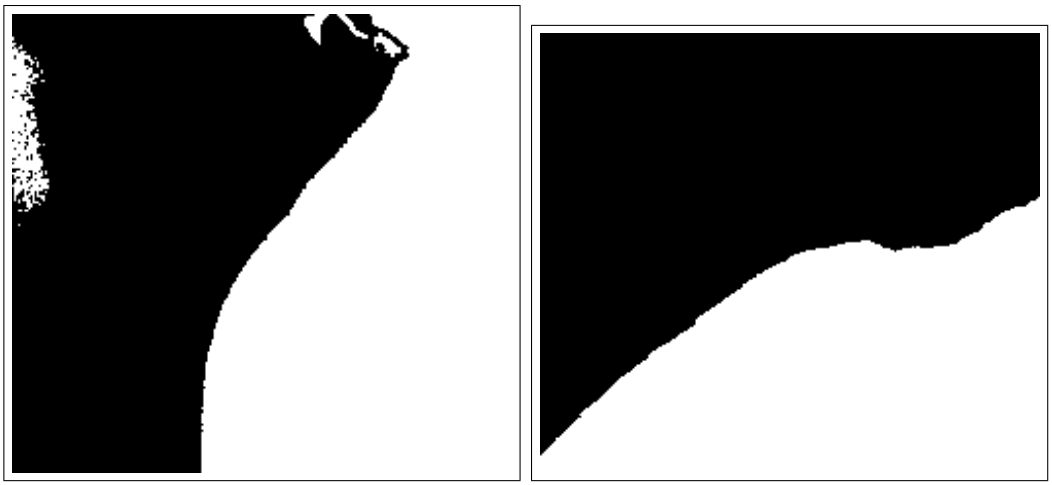

Figure 2.3: Processed images of Ejerslevlyng (left) and Gl Strandskov (right) beaches in DK with the black and white colormap applied.

2.2.4 Spark NDWI Analyzer

The Spark NDWI Analyzer is a python spark job created to continuously consume and analyze messages from Kafka to provide insight into what the scraped satellite images can tell us about the water content changes. EcoBeach relies on distributed Spark workers to handle this job, to efficiently analyze newly scraped satellite images, and feed the results back to Kafka.

The Spark NDWI Analyzer runs constantly and watches for changes to the Kafka topic "ndwi_images". Whenever a new message is added to the topic, the Spark NDWI Analyzer consumes this message, analyzes it, and produces a new message to the "ndwi_results" topic that holds all data that the data pipeline has fully processed.

The analysis is a three-step process. First the black and white images in Figure 2.3 that are created by the SSS are recreated. The images are stored in the Kafka message that contains the image encrypted as byte64. Then all white and black pixels are counted in the image, and lastly, the percentage and area (in cubic meters) of both white (water) and black (land) pixels are calculated. The relevant code is shown in Figure 2.4.

```python
def createImage(imageBytes):
    base64_imageBytes = imageBytes.encode()
    imageBytes = base64.b64decode(base64_imageBytes)
    return Image.open(io.BytesIO(imageBytes))

@udf(returnType=DoubleType())
def calculatePercentage(imageBytes, squareMeters):
    image = createImage(imageBytes)
    width, height = image.size
    return squareMeters/((width * height) * 10)

@udf(returnType=IntegerType())
def calculateWaterSquareMeters(imageBytes):
    return countPixelsFromRgb(createImage(imageBytes), (255, 255, 255, 255)) * 10

def countPixelsFromRgb(image, rgb):
    pixels = image.getdata()
    counter = Counter(pixels)
    return counter[rgb]
```

Figure 2.4: The code that recreates the black-and-white images, counts pixels and calculates percentage and area (in cubic meters) of water and land.

2.2.5 WebAPI

Our system requires a Web API to handle the data flow between the Android application and the database. This is first and foremost done to increase security and prevent any direct access to the database from the application. In our case, our application only requests data from the API, it does not add or change any items in the database, so this is a one-way dataflow.

The API accesses the MongoDB database via MongoDB Driver which is an Object–relational mapping (ORM) tool that we use to map data coming from the database to actual Beach objects which the API can handle, process and send as Json objects to endpoint-callers.

The API is deployed in a Docker container for easier testing and configurability. The database connection string can then be set in a system environment variable when launching the container. Once the container is running data can be requested on two endpoints.

/api/Beach

The application can call the API to request all the stored beach data from the database. This is first used when the application has to display the pins on the map, indicating the locations of all the beaches. The requested data is sent by the API in a JSON format, which is processed by the Android application into actual Beach objects.

/api/Beach/id

The application can also call an API request to get a single beach item from the database, based on the provided ID. This call is used when the user selects a beach in the application, and the detailed information is requested to display it for the user.

Architecture

The backend services hosting the API is a multi-tier architecture system which breaks down into three main components:

- Presentation Tier – in this system's case, it contains the hosted API which can be called externally. It contains the controllers that handle all the API requests received.
- Logic Tier – This tier mainly contains the models and helper classes. Models describe the type of object the ORM should map the data from the database. Also provides methods to read files with .csv extensions and use them to seed an empty database with beaches from the environmental agency's dataset.
- Data Tier – Provides services and context for the backend to reach the MongoDB database and queries it for data when needed.

Introducing multiple tiers in our architecture allows us to have a very decoupled system in case a tier needs to be changed, or used in another system of similar kind.

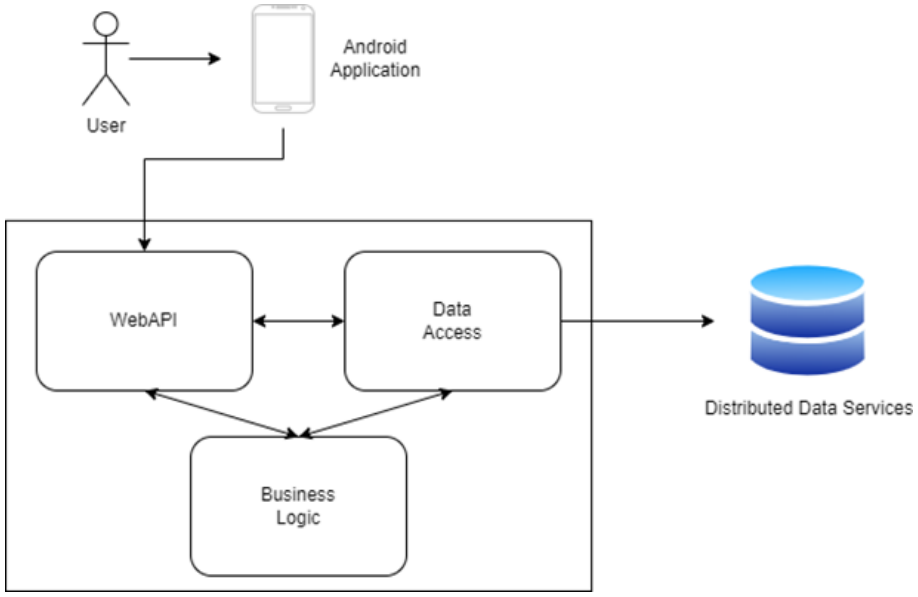

Figure 2.5: WebAPI Multi-layer architecture

2.2.6 EcoBeach App

Activities and layouts

The most important component of an application is the Activity, which is not only responsible for the user interface, but also for the operation of the application itself. During the implementation, three Activities have been created. These activities are the followings:

- SplashScreenActivity - After starting the application this is the first thing that the user can see for two seconds. On the centre of the screen there is the logo of the application and below that, there is the name of the application. We used Constraint layout, which is one of the latest and most efficient layout types.
- MainActivity - In this activity the user can see the Google Maps, with markers, that shows the different beaches in Europe. At first, it is going to focus on the user's location.
- BeachActvity - Here you can see more information about a specific beach and can see a satellite map focused on the location.

Each activity has their own layout, on the following pictures, you can see three screenshots of the different layouts.

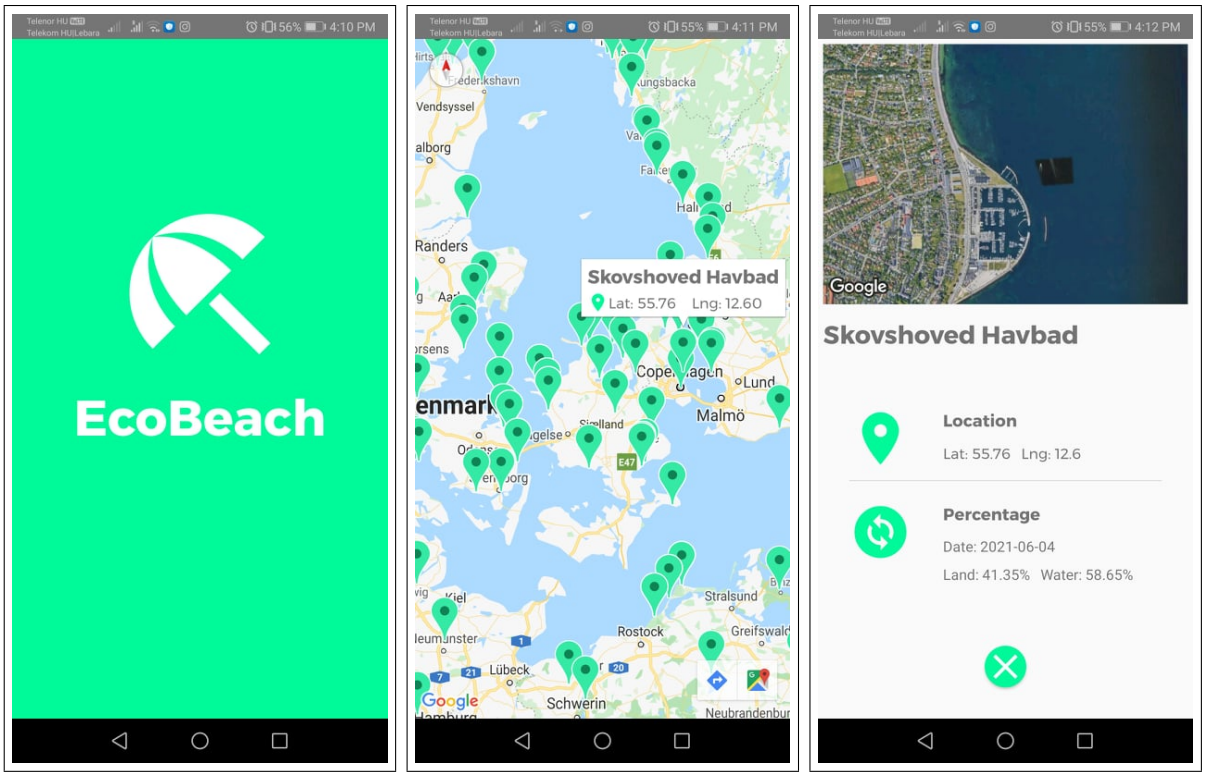

Figure 2.6: Screenshots of SplashScreenActivity (left), MainActivity (center) and BeachActivity (right).

Navigation

On Figure 2.7, the navigation between the activities is shown. After starting the application, the users can see the SplashScreen Activity for two seconds, then they are navigated to the MainActivity, which shows them the marked Google Maps. Clicking on a marked place the user is navigated to the BeachActivity, where they can be informed about the beach, like the name of the beach, the coordinates, and about the shoreline changes.

Having Internet and location enabled are very important, so in the MainActivity both are checked. If either the Internet or the location is not enabled, the user receives an alert, and the application is closed.

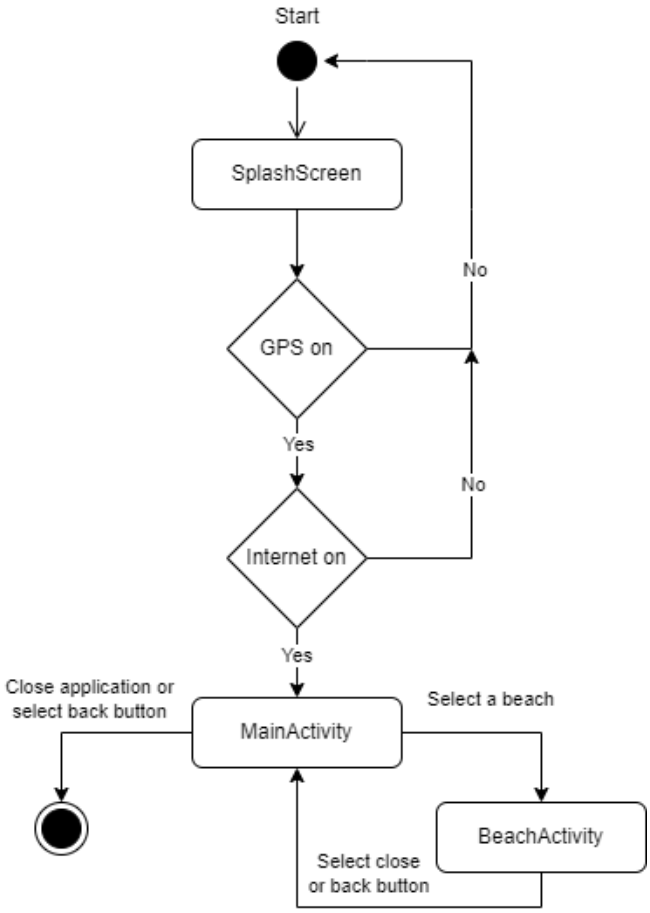

Figure 2.7: Navigation between the Activities.

User permissions

Three permissions are needed to use this application. These permissions can be seen on Figure 2.8.

```
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.INTERNET" />
```

Figure 2.8: The user must grant these permissions in order to use the EcoBeach application.

Location permissions are needed for getting the user's location. If the user is not accepting it, then the application is closed automatically, since the app is based on having the location permission.

Internet is needed for Google Maps, so that we can display different beaches on the map. Without this, we are unable to show the map in the MainActivity. However, the internet is not only used for Google Maps, but also for the Web API.

Chapter 3: Conclusion

As a solution to the enclosing threat of rising shorelines, we have developed a proof of concept system called EcoBeach. EcoBeach is a highly scalable system that monitors water content changes from satellite imagery in real-time. EcoBeach's data pipeline ensures high availability and few points of failure. Data is continuously fed to a distributed MongoDB database, so it is readily available from an Android application that can query, plot, and visualize the monitored locations in correlation to a user's location.

The data pipeline resides in a docker swarm cluster that consists of three servers, two in Germany - more specifically in Nuremberg and Falkenstein. The third server is in Helsinki, Finland, which is the name node of our cluster. Each server has a scraper that downloads and processes satellite imagery from Copernicus. The scrapers publish their data to a specific Kafka topic for each node. Apache Spark master also runs on the Helsinki node, distributing spark jobs to worker nodes to analyze the beach images. Once data has been analyzed, it is saved to another Kafka topic. A Kafka Connect Sink is set up to feed the data from this topic into our distributed MongoDB database. The database is hosted on two nodes for redundancy.

Our frontend is an Android application that runs the Google Maps API to display a map the user can navigate. Once the application is loaded, it requests data from one of the Web API endpoints that also resides in the cluster. The API then makes a data request to the database and maps the received data into one or more beach objects. The API then sends the object in a JSON format to the app, which processes it and uses the geolocation data to place each beach on the map.

The system's goal is to provide users a simple way to see how their environment changes in real-time due to climate change and raise awareness of how the sea level rises around the world.

Acronyms

NDWI Normalized Difference Water Index. 4, 14, 15

SSS Sentinel Satellite Scraper. 10, 12, 13, 14, 15, 16

Bibliography

[1] A. Cazenave and G. L. Cozannet, "Sea level rise and its coastal impacts," en, *Earth's Future*, vol. 2, no. 2, pp. 15–34, Feb. 2014, ISSN: 2328-4277, 2328-4277. DOI: 10.1002/2013EF000188. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/2013EF000188 (visited on 01/01/2022).

[2] *The Maeslantkering storm surge barrier*, en, Feb. 2013. [Online]. Available: https://www.holland.com/global/tourism/destinations/provinces/south-holland/the-maeslantkering-storm-surge-barrier.htm (visited on 01/01/2022).

[3] *Sentinel-2 product specification*, en. [Online]. Available: https://sentinel.esa.int/documents/247904/685211/Sentinel-2-Products-Specification-Document (visited on 01/01/2022).

[4] *Bathing Water Directive - Status of bathing water — European Environment Agency*, en, Data. [Online]. Available: https://www.eea.europa.eu/data-and-maps/data/bathing-water-directive-status-of-bathing-water-13 (visited on 01/01/2022).

[5] *Normalized difference water index*, en. [Online]. Available: https://en.wikipedia.org/wiki/Normalized_difference_water_index (visited on 01/01/2022).

[6] *Sentinel-2 bands combinations*, en. [Online]. Available: https://gisgeography.com/sentinel-2-bands-combinations/ (visited on 01/01/2022).

Appendix A: Infrastructure Diagram

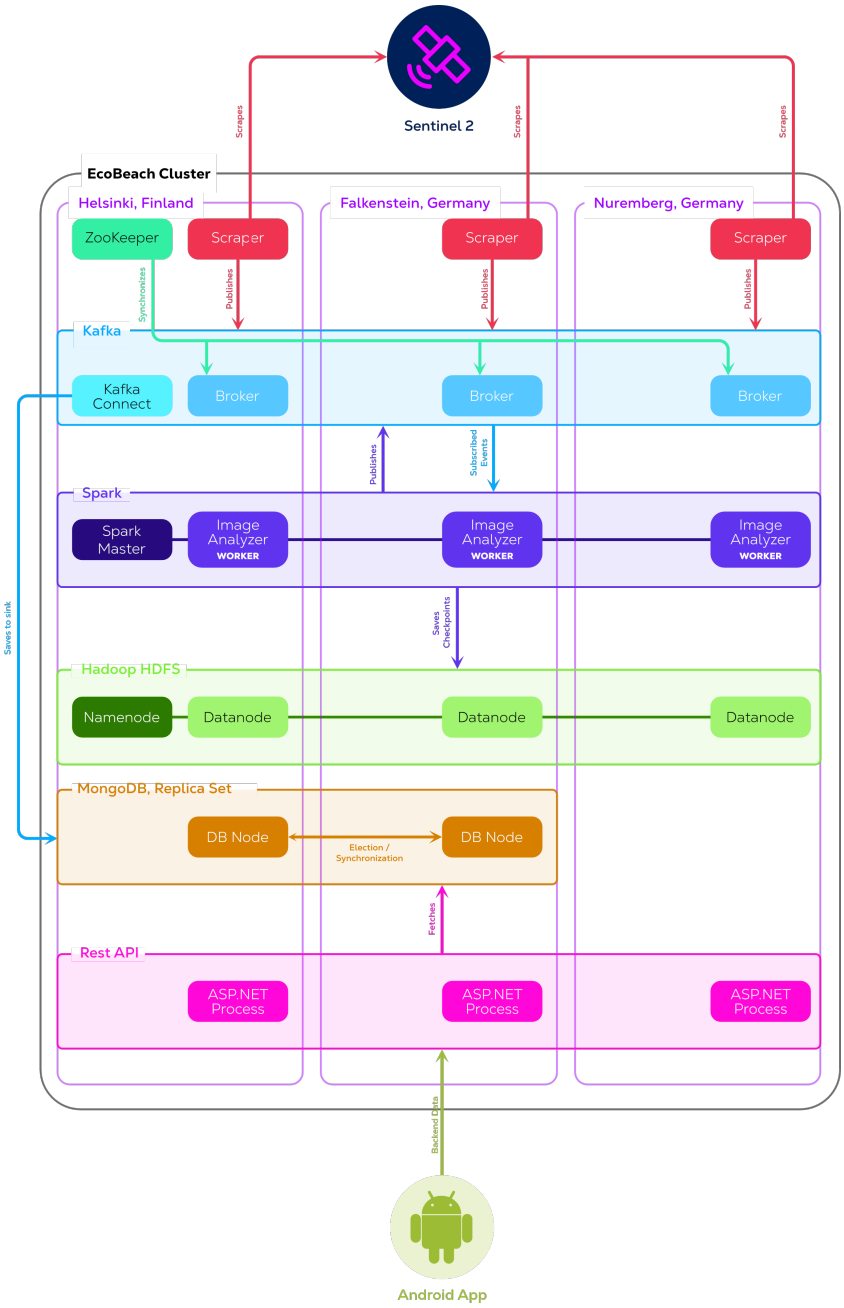

Figure A.1: The infrastructure of EcoBeach and how the included services relate to each other.