Guide Guru - Interactive Travel Guide Project Report

Tobias Kothbauer, Veronika Leitner

February 5, 2024

Abstract

In today's ever-changing world of travel, Guide Guru steps forward as a personal solution, ready to revolutionize how users embark on their journeys. Rooted in the fusion of cutting-edge technology and user-centric design philosophy, this project endeavors to craft a travel guide application that avoids limitations of traditional planning methods.

At its core, Guide Guru is driven by the vision of empowering travelers with a personalized experience, one that seamlessly aligns with their interests, preferences and hobbies. Leveraging the capabilities of advanced large language models, the application will function as an intuitive digital companion, which adapts to the desires of each user. By fostering a tight relationship between technology and user input, Guide Guru endeavors to simplify the travel planning process, offering a platform where every journey is curated to reflect the personality of the individual.

Contents

1 Aims and Context **3**

2 Project Details **4**
2.1 Questionaire 4
2.2 Technology Stack 4
2.3 Prototype Design 5
2.4 Implementation First Prototype 5
2.5 User Testing 6
2.5.1 Findings 6
2.5.2 Further Enhancements 7
2.6 Final Prototype 7
2.7 Challenges 7
2.8 Lessions Learned 8

3 System Documentation **9**
3.1 Backend 9
3.1.1 Backend API with Flask 9
3.1.2 Using the Large Language Model 10
3.1.3 Integration of Google APIs 11
3.1.4 Geonames API 11
3.2 Frontend 12
3.2.1 State Management 12
3.2.2 Pages 12
3.2.3 Components 19
3.2.4 React-PDF 19

4 Summary **22**

A Supplementary Materials **23**

References **24**

Chapter 1

Aims and Context

The core objective of this project is to develop an intuitive and adaptable travel guide application that seamlessly integrates user input with language processing capabilities. Through the implementation of user surveys, Guide Guru should gather and evaluate individual interests, ensuring that every travel recommendation is finely tuned to match the user's specific desires and hobbies.

The envisioned application gives users a tool with a simple interface, allowing them to effortlessly select destinations and specify personal interests, thereby generating travel guides curated to their liking.

Moreover, Guide Guru will offer the practical functionality of exporting personalized guides in PDF format, enabling users to conveniently access their curated recommendations on various devices and platforms.

Upon completion, Guide Guru targets delivering a travel companion, that eases the planning process and allows a high levels of customization and efficiency. By placing the user at the center of the experience, Guide Guru aims to enhance the quality of travel adventures, empowering individuals to craft marvelous journeys that align, with their interests and preferences.

In order to accurately reflect user preferences in Guide Guru, a questionnaire was created. Thirty-three participants provided insights into their travel habits, hobbies, interests, and their perspectives on the ideal format for a travel guide. This data has allowed us to customize Guide Guru to cater to a diverse audience, requiring minimal user input while still meeting their needs effectively.

Chapter 2

Project Details

Throughout our project, we carefully navigated through crucial steps to ensure success and produce great outcomes for our Guide Guru. From crafting detailed surveys to choosing the best technology tools, our path was defined by strategic choices aimed at crafting a tailored and efficient travel guide experience.

2.1 Questionaire

Initially, we developed a questionnaire based on our understanding of the essential information required to create a straightforward yet impactful prototype with highly personalized outcomes.

This survey included questions about user preferences, assessments of the importance of different travel factors, and open-ended questions to uncover insights into hobbies and preferred content for the travel guide.

Key findings from the survey revealed that the majority of participants preferred selecting their destination rather than receiving recommendations. Primary considerations for their travels included the type of travel (e.g., relaxation vacation), the overall environment (e.g., beach or city), and the preferred season for travel. Discovering new hobbies ranked as the least important aspect. All participants expressed a preference for receiving the travel guide in PDF format, with a strong emphasis on the importance of suitable images. Additionally, 85% favored selecting their hobbies from a predefined list, with sports and art & culture emerging as the most frequently mentioned categories. Regarding desired information in the travel guide, participants highlighted sights and hidden gems, culinary recommendations, entertainment options, cultural insights, and information on country-specific risks and hazards.

2.2 Technology Stack

Simultaneously, we conducted research on potential technology stacks. Two components were predetermined: utilizing React[1] for frontend development to enhance our web development skills with this technology, and selecting ChatGPT from OpenAI[2] as our

[1] https://react.dev
[2] https://openai.com

preferred large language model (LLM), thus our supervisor thankfully could provide us with an API key for this model.

Following several iterations, we opted to employ Python[3] for backend development after discovering its well-documented approach to integrating a LLM into a web application. As well Flask[4], a micro web framework, was chosen to handle requests and CORS policies.

Given the importance of images as indicated by survey participants, and considering the multitude of cities and places of interest generated by user input, we needed a means to access online data for countless locations. To achieve this, we utilized different functionalities from the Google Places API[5] to download and display images from user recommendations, providing users with a simple yet effective insight into selected destinations.

To validate user input for potential travel destinations, we integrated the geonames API[6], which is user-curated and lists millions of places, allowing filtering for highly populated areas. This API facilitated the inclusion of destinations with populations exceeding 1000, ensuring a diverse range of travel destinations could be used to generate the guide.

2.3 Prototype Design

Equipped with insights gleaned from our survey, we embarked on the development of an initial mockup for Guide Guru. Leveraging Figma[7], a collaborative design tool, we crafted a clickable prototype encompassing the entire user journey and a blueprint for our PDF guide. This mockup (see figure 2.1) enabled us to pinpoint necessary input parameters, map out pages, identify reusable components, and discover the inherent structure of our web application in a fast and efficient manner.

2.4 Implementation First Prototype

Armed with insights gathered from the prototype design step, we embarked on implementing the initial iteration of our Guide Guru prototype as an web application. This involved constructing components for various functionalities, deploying them across multiple pages, integrating diverse APIs, establishing stores to manage user input and guide output, and configuring the large language model to generate meaningful content for our guide. We conducted experiments with different setups and prompts for text generation and carefully tested the outcomes for reliability.

In crafting the user journey, we thoroughly aligned it with the principles outlined in Nielsen's 10 usability heuristics [1]. To improve our prototype even more according to these heuristics, we chose to include user testing as part of our project plan.

[3] https://www.python.org
[4] https://flask.palletsprojects.com/en/3.0.x/
[5] https://developers.google.com/maps/documentation/places/web-service?hl=de
[6] https://www.geonames.org
[7] https://www.figma.com/de

Figure 2.1: Mockup of Applicaton and Guide

2.5 User Testing

Conducting user testing with potential users is a crucial step in ensuring the quality of a prototype and identifying any potential bugs or errors that developers may overlook. By involving 5 potential users in testing our first iteration of the prototype, we aimed to gain qualitative insights into possible user journeys and identify any usability issues or challenges they encountered.

2.5.1 Findings

At the start of the test, each user received a brief introduction about what the test would cover. They were asked to imagine themselves planning a specific journey and to enunciate their thoughts on the interaction. After this initial introduction, users were not given any further instructions on how to use the prototype.

Some key findings from our user tests were that users expressed a desire to search not just for cities, but also for countries in order to select a city located within a specific country. Additionally, users noted that listing the country alongside city options could prevent errors in selection, especially when cities share the same name. The text displayed on input cards was considered insufficiently legible. On the result page, which included a sample guide, it led to confusion among users when the guide featured a different city from the one they had selected.

One user encountered a bug that disabled the large language model. This occurred when the user attempted to use the `select all` option within the selection of additional

hobbies, resulting in the prompt exceeding its maximum length and causing the backend to shut down. Another user selected a hobby that was unavailable in the destination city, prompting the language model to generate an apology stating it couldn't find relevant data, which was then displayed in the guide.

2.5.2 Further Enhancements

Two of the five test user would like to include an duration of the vacation an thus an suitable amount of places of interest for that duration. Three users thought of ideas to avoid the generation time with facts or images about the destination or an pageful output of the guide. Only one thought that the travel guide contained incorrect information for the places of interest. The user thought that probably two niche hobbies would not be part of an tournament in the same park.

Two out of the five test users expressed interest in including the duration of the vacation to determine a suitable number of places of interest for that period. Three users brainstormed ideas to mitigate generation time, such as incorporating facts or images about the destination or showing the result of each fully generated page as soon as it was available. Only one user thought that the travel guide provided incorrect information regarding the places of interest. The guide stated that two niche hobbies have independent tournaments in the same park. Further research didn't provide any information about the tournaments in this park so the description of the place was presumably wrong.

2.6 Final Prototype

In the final prototype, we addressed several bugs and incorporated additional features to unleash the full potential of Guide Guru. For instance, we included countries to the destinations and enhanced the selection process by offering initial options consisting of cities with populations exceeding 15,000 residents. We overhaul the style of card selection, improved responsiveness, and implemented form validation requiring a minimum selection. One of the most significant updates occurred on the result page, where we eliminated the example guide and introduced a table displaying previous user inputs, allowing for easy modifications if necessary. Additionally, we integrated an interactive map to engage users while they await the generation of the final guide.

2.7 Challenges

One of the major challenges we encountered in this project was setting up the backend for the large language model. This process involved multiple iterative steps, and debugging issues through error logs was complex due to various dependencies. Additionally, since the setup was local, both team members needed a valid backend setup to continue working on the project.

Once the LLM backend was established, another challenge arose: generating output that met the requirements for the descriptions of places, as well as formatting the numbers for country and city area and citizens correctly. This required fine-tuning

prompts for ChatGPT and generating multiple iterations of possible user input to validate functionality. While we couldn't verify functionality for every input, extensive testing, including user testing, gave us confidence that it works for most touristically relevant places with sufficient data available.

One of the challenges we are still grappling with is displaying the generated travel guide in a new tab to ensuring users can navigate back to Guide Guru without losing the guide if it's not downloaded. Our current setup saves the generated output into our redux storage, making it inaccessible in another tab. When opened in a new tab, the information can not be displayed. Attempting to open a new tab with a new request also results in data loss, disrupting the user journey and preventing users from tracking the system's status.

We have not implemented a solution yet, but we have two ideas. One is to use local storage instead of the redux store. Another option is to store user input or the LLM output in a database. However, as our project deadline was approaching when we came up with these ideas, we were not able to implement them.

Most of the other challenges we faced during prototype implementation involved fixing small bugs and dealing with unpredictable outputs.

2.8 Lessions Learned

The main lessons we learned during our project were closely tied to the challenges we encountered. We learned how to set up a Python backend and engage in prompt engineering to ensure sufficient ChatGPT output. Working with the somewhat unpredictable output of the LLM proved to be a significant learning. While the LLM provided quick results, generating reliable output tailored to specific needs was challenging, and fine-tuning the model required a considerable amount of time.

Additionally, we gained valuable knowledge about React as a frontend technology. We learned effective methods for separating content and testing the web application thoroughly.

These experiences broadened our understanding of both backend and frontend development processes, highlighting the importance of careful planning and iteration to achieve desired outcomes.

Chapter 3

System Documentation

3.1 Backend

3.1.1 Backend API with Flask

In this section, we will establish the backend API using Flask. Flask is a prominent Python web framework utilized for building RESTful APIs. We chose this framework because of its simplicity and flexibility. Our goal was to create a backend infrastructure that integrates Flask and ChatGPT functionalities. The API enables communication between the user interface and the AI model, ensuring efficient query handling and response generation. The Flask API looks like the following:

```
class ChatHandler(Resource):
def __init__(self, **kwargs):
self.context = [{"role": "system", "content": system_context}]

def get(self):
response = jsonify(context=self.context)
response.headers.add("Access-Control-Allow-Origin", "*")
return response

def post(self):
rqs = request.json
print(f'POST Print RQS: {rqs}\n')
destination = rqs.get('destination')
seasons = rqs.get('seasons')
travelType = rqs.get('travelType')
hobbies = rqs.get('hobbies')
addHobbies = rqs.get('addHobbies')
country = rqs.get('country')

print(f'POST: {destination}\n{country}\n')

response = jsonify(get_response(destination, hobbies, travelType, seasons,
  addHobbies, country))
return response


api.add_resource(ChatHandler, '/')
```

The API handles HTTP POST requests which are made to the endpoint '/'. The post method retrievs the sent JSON and extracts relevant information such as destination, seasons, travel type, hobbies, additional hobbies, and country. The extracted data is then passed to the function get_response() from the utility module which processes these user's preferences and will generate a suitable response with information about the destination and personalized recommendations

3.1.2 Using the Large Language Model

In the utility.py module, we utilize the openai Python package to streamline our requests. Initially, we load the OpenAI API key stored in the .env file located in the root directory. The get_response method is designed to retrieve personalized travel recommendations based on the parameters provided by the user. Initially the OpenAI model is initialized with the "gpt-3.5-turbo" model.

```
def get_response(
destination: str, hobbies: list, travelType: list, seasons: list, addHobbies: list
  , country: str):
gpt = models.OpenAI("gpt-3.5-turbo", api_key=api_key)
```

We utilize the guidance library[1] for our prompting process. This programming paradigm provides enhanced control and efficiency compared to traditional prompting and chaining methods. For the prompting we are using the different roles "system", "user" and "assistant". This is crucial because we require a separate prompt for each aspect, which we then save and send back to the frontend. First of all we need to define the system context for the model:

```
with system():
lm = gpt + "You act as an AI that will create a personalized Travel Guide for a
  User. You are a trained bot with the best personalized travel suggestions based
  on the users input. You analyze the inputfrom the user and decide based on that.
   At the end you will provide a Travel Guide with a specific location and
  suggestions based on the user needs."
```

After that we created step by step the wanted prompts. In the following example we ask ChatGPT to search for 5 places based on the information the user provided. We also limited the text to 100 words:

```
with user():
lm += "Please search for 5 places to go in " + destination + " within a " +
  travelTypeString + " vacation while the " + seasonString + " seasons, with a
  description with around 100 words."

with assistant():
lm += gen("answerPlaces")
places_text = lm["answerPlaces"]
print(f'places_text: {places_text}')
```

After the assistant answers our prompt, we store it in the "places_text" variable, from where we can use it.

We also used ChatGPT to get the currency, area and population of the city as well as for the associated country. For that we changed the model to "gpt-4", because we received better results than with the "gpt-3.5-turbo" model.

[1] https://github.com/guidance-ai/guidance

After all prompts have been created and we have saved the corresponding answers, we just return them which will then be the response of the API, if the post method is called 3.1.1.

3.1.3 Integration of Google APIs

We choose the Google Maps API to retrieve photos of specific locations based on textual queries. The text_search function queries places based on user input by using the Google Places API. If a suitable place is found, we can use the "place_id" to retrieve various details like the "photo_reference". With this information we are able to assemble the photo URLs and save it in an array.

```
def text_search(query, max_width=4000, max_height=4000):
places = gmaps.places(query=query)
place_id = places['results'][0]['place_id']
place_details = gmaps.place(place_id=place_id)
photo_urls = []
for photo_info in place_details['result']['photos']:
photo_reference = photo_info['photo_reference']
photo_url = f'https://maps.googleapis.com/maps/api/place/photo?
maxwidth={max_width}&maxheight={max_height}&photoreference={
  photo_reference}&key={API_KEY}'
photo_urls.append(photo_url)
return photo_urls
```

Following that, we established the download_images function. This function is designed to accept an array of photo URLs and a designated folder location as parameters. Its purpose is to download and save the images from the provided array to the specified location.

3.1.4 Geonames API

We utilized the geonames API to validate user input while keeping the range of possible destinations open. The geonames API contains data on approximately 4.8 million populated places, including details such as name, alternative names, country, population, geolocation, and categorization. To refine the user options, we set a threshold of including only places with over 1,000 inhabitants, ensuring that the travel guide's output remains relevant and accurate.

For the integration of this API into our Guide Guru, we used Express.js[2] to create the API Interface. After incorporating cors and axios middleware, we developed two interfaces for communicating with the geonames API.

To enhance user experience, we designed a mechanism to display only well-known and populous destinations upon the initial opening of the dropdown field. Upon rendering the destination page, a request is triggered, prompting an axios-request to the geonames API which can be seen in the below code. This request retrieves 25 cities with over 15,000 residents in its response, ensuring that users are presented with familiar and prominent locations.

```
app.get('/api/bigCity', (req, res) => {
```

[2] https://expressjs.com/de/

```
  const apiUrl = 'http://api.geonames.org/searchJSON?cities=cities15000&maxRows=25&
    username=fhtravelguidews23';
  axios.get(apiUrl)
  .then(response => {
    res.json(response.data)
  })
  .catch(error => {
    console.error('Error fetching data:', error);
  });
});
```

As soon as the user begins to enter into the input field an request is sended for ten places that match the input an have more than 1,000 citicens to allow a high variety of possible travel destinations.

Once the user starts typing in the input field, a request is sent to retrieve ten places that match the input criteria and have more than 1,000 inhabitants, thus providing a diverse range of potential travel destinations.

The processing of the API response data is detailed in the description of our Destination Page at 3.2.2.

3.2 Frontend

As mentioned previously, we opted for React as our frontend technology. To maintain a component-based architecture, we divided our frontend into pages that leverage multiple components we designed and implemented. Additionally, we established a state management system to store both user input and data provided by the backend. This mixture of technologies enables us to create a straightforward yet efficient user journey.

3.2.1 State Management

For state management, we utilized the Redux library to store both user input and data retrieved from the backend. The `inputSlice.js` file is responsible for storing the input of the user regarding the desired destination, seasons, type of travel, and selected hobbies they want to include in their journey. Additionally, we implemented a function within this slice to reset the user's input for new requests without requiring a page reload.

Within the `guideSlice.js` file, all information necessary to construct the guide is saved. This includes details such as destination, population, area, longitude, latitude, country information, images with appropriate dimensions, and recommended places.

In the file `store.ts`, these two reducers are configured and made accessible within the store variable. For instance, `store.getState().userInput.destination` provides access to the user's selected destination. To add data into the store, you can use functions like `dispatch(setDestionation("destination"))`.

3.2.2 Pages

In our mockup design, we implemented separate pages for each stage of the user journey. Each page features the header component, providing a consistent element throughout the experience for user familiarity. To ensure the validity of every guide request, we

Figure 3.1: Homepage of Guide Guru

opted against traditional navigation methods and instead enabled users to navigate solely using buttons located at the bottom of each page.

Homepage

The homepage serves as the initial step in the user journey. To facilitate a seamless start to their personalized experience, we included our logo and a brief introduction explaining how to use the application. Additionally, to encourage continued engagement with the website, we incorporated a call to action featuring a button labeled *Start your journey*, directing users to the first input page.

Destination

In the destination page, users are prompted to choose their preferred travel destination from options provided by our geonames API interface. Upon the initial rendering of the page, a request is made for 25 cities with more than 15,000 inhabitants. The responses from this interface populate the options, where the city name serves as the value and part of the label, along with the country name to prevent confusion between places with similar names. Additionally, information such as city population, country, latitude, and

longitude is included in the options to be stored in the state when the user selects a destination. We utilized the `react-select` library to build this selection component, and the required HTML structure is as follows:

```
  <form onSubmit={formSubmit}>
    <div className="my-4 flex justify-center">
      <Select
        isSearchable={true}
        options={cityOptions}
        onInputChange={loadOptions}
        required={true}
        onChange={handleChange}
        placeholder="start typing or select a city ..."
      />
    </div>
    <Button text='Continue' type='submit'/>
  </form>
```

To send requests to the interface and build the adapted options as described above when the user starts typing into the input field, we utilize the following TypeScript function. This function is called from the `onInputChange` event of the `<Select>` component:

```
const [cityData, setCityData] = useState([]);
const [cityOptions, setCityOptions] = useState([]);
const dispatch = useDispatch();

const loadOptions = (inputValue: string) => {
  axios.get('http://127.0.0.1:4000/api/city?city=' + inputValue, {}).then((response)
     => {
    setCityData(response.data.geonames);
  }).catch(error => console.log(error));
};

useEffect(() => {
  createOptions();
}, [cityData]);

function createOptions() {
  const newOptions = []
  cityData.forEach((city) => {
    newOptions.push({
      ...city.name,
      value: city.name,
      label: city.name + ' (' + city.countryName + ')',
      population : city.population,
      country: city.countryName,
      lat: city.lat,
      lng: city.lng,
    })
  })
  setCityOptions(newOptions);
}

const handleChange = (selectedOption) => {
  dispatch(setDestination(selectedOption.value));
  let cityCiticens = selectedOption.population.toString().replace(/\B(?=(\d{3})+(?!\
```

```
      d))/g, ",");
  dispatch(setCitizens(cityCiticens));
  dispatch(setCountry(selectedOption.country));
  dispatch(setLatitude(selectedOption.lat));
  dispatch(setLongitude(selectedOption.lng));
};
```

Within the `loadOptions` function, a request is made to the interface, and the hook for `cityData` is adapted accordingly. This change triggers the `useEffect`, which constructs the new options for the selection, refreshing the `cityOptions` hook.

Once the user selects a destination, the `handleChange` function is accessed. This function adds the necessary and formatted information into the stores. Therefore, the required data is already saved, and the `onSubmit` function simply redirects to the next page.

Seasons

On the season page, users are required to select at least one of the four seasons. The selection methodology is described within the `CardBlock` component at 3.2.3. To ensure that the user has selected at least one season, the form submission process checks the season store for elements. If the season store is empty, the message indicating the selection requirements switches from the regular color to bold red as can be seen in the HTML and TypeScript code below.

```
<form onSubmit={handleSubmit(onSubmit)}>
  <Heading headingLevel='h3' text="In which season would you like to travel?"/>
  <Paragraph text="Select one or multiple seasons for your journey."
    className={`${textColor} ${isBold ? 'font-bold' : ''}`}/>
  <CardBlock category="seasons" name="seasons"/>
</form>
```

```
const navigate = useNavigate();
const [textColor, setTextColor] = useState('text-black');
const [isBold, setIsBold] = useState(false);

function formSubmit(data: UserInput) {
  if (store.getState().userInput.seasons.length === 0) {
    setTextColor('text-error');
    setIsBold(true);
  } else {
    navigate('/type');
  }
}

const onSubmit: SubmitHandler<UserInput> = data => formSubmit(data);
```

Once the requirements are met, the user is redirected to the travel type page.

Travel Type

The functionality of the travel type page mirrors that of the seasons page. Users must select at least one element in the `CardBlock` component to ensure a correct output from the large language model. Additionally, this requirement aims to motivate users to enter a category, which will be used to tailor the guide to their individual needs.

Figure 3.2: Selection of the type of travel

Hobbies

The hobby page shares similarities with the previous pages, but includes three different `CardBlock` components, one of which contains the `HobbySelection` component described at 3.2.3.

To ensure tailored and appropriate travel guide outputs, we've increased the minimum selected options to three hobbies, either inside the `CardBlock` or the `HobbySelection` component. This adjustment aims to prevent undefined or incorrect outputs for points of interest when selected hobbies are not possible at the chosen destination (e.g., skiing in Hawaii). By requiring at least three hobbies to be selected, we anticipate that the guide will provide some relevant points of interest.

The functionality of this page remains largely the same, but includes a small computation for the `CardBlock` and the additional selected hobbies in the `HobbySelection` component, as demonstrated below.

```
const [textColor, setTextColor] = useState('text-black');
const [isBold, setIsBold] = useState(false);

```

```
function formSubmit(data: UserInput) {
  const totalHobbies = store.getState().userInput.hobbies.length +
    store.getState().userInput.addHobbies.length;
  if (totalHobbies < 3) {
    setTextColor('text-error');
    setIsBold(true);
  } else {
    navigate('/result');
  }
}
```

Result

At the result page, users are presented with their personal input. We've provided users with the option to validate their input before generating the guide. This is important because the guide generation process can take one to two minutes of computation time, and users should have the possibility to establish it when they are pleased with their input. On this page, users can navigate to each of the previous input pages and change their selection or start a new travel guide with new parameters.

When the user clicks on the generate button, a request with the stored input data is sent to the backend to be computed by the large language model. The resulting guide is then stored in the `guideStore` to be accessed in the generated PDF document (see 3.2.4), to which the user is subsequently redirected.

In the code below the request to the backend to generate the guide and the handling of the output is shown:

```
const fetchChatResponse = () => {
  setIsLoading(true); // shows the loading animation, interactive map and message

  // load the stored inputs
  const destination = store.getState().userInput.destination;
  const seasons = store.getState().userInput.seasons;
  const travelType = store.getState().userInput.travelType;
  const hobbies = store.getState().userInput.hobbies;
  const addHobbies = store.getState().userInput.addHobbies;
  const country = store.getState().guide.country.name;

  // request to the LLM
  axios.post('http://127.0.0.1:5000/', {
    destination: destination,
    seasons: seasons,
    travelType: travelType,
    hobbies: hobbies,
    addHobbies: addHobbies,
    country: country
  }).then((response) => {
    // save data into guideStore
    dispatch(setDescription(response?.data.destination_text));
    dispatch(setDestination(destination));
    dispatch(setArea(shortenAnswerForNumbers(response?.data.city_area)));
    dispatch(setCountryCitizens(shortenAnswerForNumbers(response?.
      data.country_citicens)));
    dispatch(setCountryArea(shortenAnswerForNumbers(response?.
      data.country_area)));
```

Figure 3.3: Result page before generation

Figure 3.4: Result page during generation

```
      dispatch(setCurrency(response?.data.currency));
      let places = response?.data.places_text;
      let placesArr = places.split(/: |\n\n/);
      dispatch(setPlacesToGo(placesArr))
      places = response?.data.poi_text;
      placesArr = places.split(/: |\n\n/);
      dispatch(setPOI(placesArr));
      setIsLoading(false);
      navigate('/exportGuide')
    }).catch(error => console.log(error));
  };
```

During the guide generation process, the user is notified that it may take some time. Additionally, an interactive map component (see 3.2.3) is displayed, showing the selected destination marked with data retrieved from the geonames API (described at 3.1.4). This allows the user to explore their chosen travel destination while waiting for the guide to be generated.

Export Guide

This page is only needed to include the PDF Viewer and the React PDF that is described in the subsection 3.2.4.

3.2.3 Components

Button

CardBlock

CardInput

Header

Heading

HobbySelection

InteractiveMap

Paragraph

3.2.4 React-PDF

React-PDF renderer is a library utilized for generating PDF files with a markup structure similar to HTML and CSS. This enables easy integration into any application. To enable the browser to open the generated PDF, a `<PDFViewer>` element is required, within which a `<Document>` is created. Then, as many `<Pages>` as necessary can be added to the document. To structure the layout of the PDF, the `<View>` element can be utilized. For including text and images, the appropriate `<Text>` and `<Image src="path/to/image">` elements can be used. With these components, we were able to create the simple yet effective layout of our travel guide, as depicted in 3.5.

For styling the PDF, it is possible to utilize the `style` attribute on every component of the React-PDF library. To enhance code readability and separation of concerns, we decided to store our styling in a separate variable. In the code snipped below the definition of the style elements used in the code in 3.6 can be seen:

Figure 3.5: Front page of the travel guide

```
<PDFViewer style={styles.viewer}>
  <Document>
    <Page size="A4" style={styles.page}>
      <div style={styles.pageAccent}></div>
        <View style={styles.section}>
          <Text style={styles.title}>Top 5</Text>
          <Text style={styles.handwritten}>Places to go</Text>
          <Image src="../src/img/pdf/imgs/image_3.jpg"
            style={styles.largeImg}></Image>
          <Image src="../src/img/pdf/icons/Icon1.png" style={[styles.numIcon,
            { position: 'absolute', top: 150, left:50}]}></Image>
          <View style={styles.section}>
          <Text style={styles.headingLg}>{placesToGo.first.name}</Text>
          <Text style={styles.description}>{placesToGo.first.description}</Text>
        </View>
      </View>
    </Page>
  </Document>
</PDFViewer>

```

```
const styles = StyleSheet.create({
  viewer: {
    width: window.innerWidth,
    height: window.innerHeight,
  },
  page: {
    backgroundColor: 'rgba(255,255,255,0.6)',
    color: "white",
    fontFamily: 'Jost',
  },
  pageAccent: {
    width: 500,
    height: 20,
    position: "absolute",
    right: 0,
    backgroundColor: '#1B7EA6'
  },
  title: {
    color: '#1B7EA6',
    fontSize: 45,
    textAlign: 'center',
    fontFamily: 'Jost',
    fontWeight: "bold"
  },
  section: {
    margin: 10,
    padding: 10,
    color: 'black',
  },
  headingLg: {
    width: ImageDimensions.large.width,
    fontSize: 20,
    color: '#1B7EA6',
  },
  handwritten: {
    fontFamily: 'Handwritten',
    textAlign: "center",
    color: '#232323',
    fontSize: 28,
    position: "absolute",
    top: 63,
    left: 250,
  },
  largeImg:{
    top: 22,
    height: ImageDimensions.large.height,
    marginBottom: 10,
    objectFit: "cover",
  },
  numIcon:{
    height: ImageDimensions.icon.height,
    width: ImageDimensions.icon.width,
    alignSelf:"center",
    marginBottom: 30,
  },
  description:{
    fontSize: 13,
    fontWeight: "light",
  }
});
```

Figure 3.6: Styling of page depicted at 3.5

Chapter 4

Summary

Give a concise (and honest) summary of what has been accomplished and what not. Point out issues that may warrant further investigation.

Appendix A

Supplementary Materials

The appendix is a good place to attach a user guide, screenshots, installation instructions, etc. Add a separate chapter for each major item.

References

[1] Jakob Nielsen. *10 Usability Heuristics for User Interface Design*. 1994. URL: https://www.nngroup.com/articles/ten-usability-heuristics/ (visited on 02/03/2024) (cit. on p. 5).